



畅销书全新升级和大幅优化，第1版广受好评，被翻译为繁体中文和英文（美国）
以真实操作系统的实际运行为主线，以图形图像为核心，突出描述操作系统在实际运行过程中内存的运行结构；从操作系统设计者的视角，用体系的思想方法，深刻解读操作系统的架构设计与实现原理



第2版

Linux 内核设计 的艺术

图解Linux操作系统架构
设计与实现原理

The Art of Linux
Kernel Design, Second Edition

新设计团队 著



机械工业出版社
China Machine Press

Linux 内核设计的艺术：图解Linux 操作系统架构设计与实现原理

新设计团队 著

ISBN: 978-7-111-42176-4

本书纸版由机械工业出版社于2013年出版，电子版由华章分社（北京华章图文信息有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @研发书局

腾讯微博 @yanfabook

目 录

前言

为什么写这本书

第1版与第2版的区别

本书内容及特色

为什么本书选用Linux 0.11内核

致谢

第1章 从开机加电到执行main函数之前的过程

1.1 启动BIOS，准备实模式下的中断向量表和中断服务程序

1.1.1 BIOS的启动原理

1.1.2 BIOS在内存中加载中断向量表和中断服务程序

1.2 加载操作系统内核程序并为保护模式做准备

1.2.1 加载第一部分内核代码——引导程序
(bootsect)

1.2.2 加载第二部分内核代码——setup

1.2.3 加载第三部分内核代码——system模
块

1.3 开始向32位模式转变，为main函数的调用
做准备

1.3.1 关中断并将system移动到内存地址起
始位置0x00000

1.3.2 设置中断描述符表和全局描述符表

1.3.3 打开A20，实现32位寻址

1.3.4 为保护模式下执行head.s做准备

1.3.5 head.s开始执行

1.4 本章小结

第2章 设备环境初始化及激活进程0

2.1 设置根设备

2.2 规划物理内存格局，设置缓冲区、虚拟盘、主内存

2.3 设置虚拟盘空间并初始化

2.4 内存管理结构mem_map初始化

2.5 异常处理类中断服务程序挂接

2.6 初始化块设备请求项结构

2.7 与建立人机交互界面相关的外设的中断服务程序挂接

2.7.1 对串行口进行设置

2.7.2 对显示器进行设置

2.7.3 对键盘进行设置

2.8 开机启动时间设置

2.9 初始化进程0

2.9.1 初始化进程0

2.9.2 设置时钟中断

2.9.3 设置系统调用总入口

2.10 初始化缓冲区管理结构

2.11 初始化硬盘

2.12 初始化软盘

2.13 开启中断

2.14 进程0由0特权级翻转到3特权级，成为真正的进程

2.15 本章小结

第3章 进程1的创建及执行

3.1 进程1的创建

3.1.1 进程0创建进程1

3.1.2 在task[64]中为进程1申请一个空闲位置并获取进程号

3.1.3 调用copy_process函数

3.1.4 设置进程1的分页管理

3.1.5 进程1共享进程0的文件

3.1.6 设置进程1在GDT中的表项

3.1.7 进程1处于就绪态

3.2 内核第一次做进程调度

3.3 轮转到进程1执行

3.3.1 进程1为安装硬盘文件系统做准备

3.3.2 进程1格式化虚拟盘并更换根设备为虚拟盘

3.3.3 进程1在根设备上加载根文件系统

3.4 本章小结

第4章 进程2的创建及执行

4.1 打开终端设备文件及复制文件句柄

4.1.1 打开标准输入设备文件

4.1.2 打开标准输出、标准错误输出设备文件

4.2 进程1创建进程2并切换到进程2执行

4.3 加载shell程序

4.3.1 关闭标准输入设备文件，打开rc文件

4.3.2 检测shell文件

4.3.3 为shell程序的执行做准备

4.3.4 执行shell程序

4.4 系统实现怠速

4.4.1 创建update进程

4.4.2 切换到shell进程执行

4.4.3 重建shell

4.5 本章小结

第5章 文件操作

5.1 安装文件系统

5.1.1 获取外设的超级块

5.1.2 确定根文件系统的挂接点

5.1.3 将超级块与根文件系统挂接

5.2 打开文件

5.2.1 将进程的*filp[20]与file_table[64]挂接

5.2.2 获取文件i节点

5.2.3 将文件i节点与file_table[64]挂接

5.3 读文件

5.3.1 确定数据块在外设中的位置

5.3.2 将数据块读入缓冲块

5.3.3 将缓冲块中的数据复制到进程空间

5.4 新建文件

5.4.1 查找文件

5.4.2 新建文件i节点

5.4.3 新建文件目录项

5.5 写文件

5.5.1 确定文件的写入位置

5.5.2 申请缓冲块

5.5.3 将指定的数据从进程空间复制到缓冲块

5.5.4 数据同步到外设的两种方法

5.6 修改文件

5.6.1 重定位文件的当前操作指针

5.6.2 修改文件

5.7 关闭文件

5.7.1 当前进程的filp与file_table[64]脱钩

5.7.2 文件i节点被释放

5.8 删除文件

5.8.1 对文件的删除条件进行检查

5.8.2 进行具体的删除工作

5.9 本章小结

第6章 用户进程与内存管理

6.1 线性地址的保护

6.1.1 进程线性地址空间的格局

6.1.2 段基址、段限长、GDT、LDT、特权级

6.2 分页

6.2.1 线性地址映射到物理地址

6.2.2 进程执行时分页

6.2.3 进程共享页面

6.2.4 内核分页

6.3 一个用户进程从创建到退出的完整过程

6.3.1 创建str1进程

6.3.2 str1进程加载的准备工作

6.3.3 str1进程的运行、加载

6.3.4 str1进程的退出

6.4 多个用户进程同时运行

6.4.1 进程调度

6.4.2 页写保护

6.5 本章小结

第7章 缓冲区和多进程操作文件

7.1 缓冲区的作用

7.2 缓冲区的总体结构

7.3 b_dev、b_blocknr及request的作用

7.3.1 保证进程与缓冲块数据交互的正确性

7.3.2 让数据在缓冲区中停留的时间尽可能长

7.4 uptodate和dirty的作用

7.4.1 b_uptodate的作用

7.4.2 b_dirty的作用

7.4.3 i_uptodate、i_dirt和s_dirt的作用

7.5 count、lock、wait、request的作用

7.5.1 b_count的作用

7.5.2 i_count的作用

7.5.3 b_lock、*b_wait的作用

7.5.4 i_lock、i_wait、s_lock、*s_wait的作用

7.5.5 补充request的作用

7.6 实例1：关于缓冲块的进程等待队列

7.7 总体来看缓冲块和请求项

7.8 实例2：多进程操作文件的综合实例

7.9 本章小结

第8章 进程间通信

8.1 管道机制

8.1.1 管道的创建过程

8.1.2 管道的操作

8.2 信号机制

8.2.1 信号的使用

8.2.2 信号对进程执行状态的影响

8.3 本章小结

第9章 操作系统的设计指导思想

9.1 运行一个最简单的程序，看操作系统为程序运行做了哪些工作

9.2 操作系统的设计指导思想——主奴机制

9.2.1 主奴机制中的进程及进程创建机制

9.2.2 操作系统的设计如何体现主奴机制

9.3 实现主奴机制的三种关键技术

9.3.1 保护和分页

9.3.2 特权级

9.3.3 中断

9.4 建立主奴机制的决定性因素——先机

9.5 软件和硬件的关系

9.5.1 非用户进程——进程0、进程1、shell 进程

9.5.2 文件与数据存储

9.6 父子进程共享页面

9.7 操作系统的全局中断与进程的局部中断 ——信号

9.8 本章小结

结束语

“新设计团队”简介

作者的话

联系作者

前言

为什么写这本书

很早就有一个想法，做中国人自己的、有所突破、有所创新的操作系统、计算机语言及编译平台。

我带领的“新设计团队”（主要由中国科学院研究生院毕业的学生组成）在实际开发自己的操作系统的过程中，最先遇到的问题就是如何培养学生真正看懂Linux操作系统的源代码的能力。开源的Linux操作系统的源代码很容易找到，但很快就会发现，培养学生看懂Linux操作系统的源代码是一件非常困难的事。

操作系统的代码量通常都是非常庞大的，动辄几百万行，即使浏览一遍也要很长时间。比庞大的代码量更让学习者绝望的是操作系统有着极其错综复杂的关系。看上去，代码的执行序时隐时现，很难抓住脉络。代码之间相互牵扯，相互勾连，几乎无法理出头绪，更谈不上理解代码背后的原理、意图和思想。

对于学生而言，选择从源代码的什么地方开始分析，本身就是一个难题。通常，学生有两种选择：一种是从main函数，也就是从C语言代码的总入口开始，沿着源代码的调用路线一行一行地看下去，学生很快就会发现源代码的调用路线莫名其妙地断了，但直觉和常识告诉他操作系统肯定不会在这个地方停止，一定还在继续运行，

却不知道后续的代码在哪里，这种方法很快就走进了死胡同；另一种则是从某一模块入手，如文件系统，但这样会无形中切断操作系统源码之间复杂的关系，如文件系统与进程管理的关系，文件系统与内存管理的关系，等等。学生如果孤立地去理解一个模块，往往只能记住一些名词和简单概念，难以真正理解操作系统的全貌。用学生的话讲，他们理解的操作系统变成了“文科”的操作系统。

由于操作系统是底层系统程序，对应用程序行之有效的调试和跟踪等手段对操作系统的源代码而言，几乎无效。学生就算把每一行源代码都看懂了，对源代码已经烂熟于心，知道这一行是一个for循环，那一行是一个调用.....但仍然不知

道整个代码究竟在做什么以及起到什么作用，更不知道设计者的意图究竟是什么。

学生在操作系统课程上学习过进程管理、内存管理、文件系统等基础知识，但是对这些空洞的理论在一个实际的操作系统中是如何实现的却不得而知。他们在源代码中很难看出进程和内存之间有什么关联，内核程序 and 用户程序有什么区别，为什么要有这些区别；也很难从源代码中看清楚，我们实际经常用到的操作，比如打开文件，操作系统在其中都做了哪些具体的工作。想在与常见的应用程序的编程方法有巨大差异的、晦涩难懂的、浩瀚如海的操作系统底层源代码中找到这些问题的答案，似乎比登天还难。

对熟悉操作系统源代码的学生而言，他们也知道像分页机制这样的知识点，知道若干级的分页及恒等映射，但是未必能够真正理解隐藏在机制背后的深刻意义。

这些都是学生在学习Linux操作系统源代码时遇到的实际问题。中国科学院研究生院的学生应该是年轻人中的佼佼者，他们遇到的问题可能其他读者也会遇到。我萌发了一个想法，虽然学生的问题早已解决，但是否可以把他们曾经在学习、研发操作系统的过程中遇到的问题和心得体会拿出来供广大读者分享。

当时，针对学生的实际问题，我的解决方法是以一个真实的操作系统为例，让学生理解源代

码并把操作系统在内存中的运行时状态画出图来。实践证明，这个方法简单有效。

现在我们把这个解决方案体现在这本书中。这就是以一个真实的操作系统的实际运行为主线；以图形、图像为核心，突出描述操作系统在实际运行过程中内存的运行时结构；强调学生站在操作系统设计者的视角，用体系的思想方法，整体把握操作系统的行为、作用、目的和意义。

第1版与第2版的区别

第2版较第1版有较大的改动。

从总体结构上，将第1版的第2章拆分为第2版的第2章、第3章、第4章。这样的拆分对操作系统启动部分的系统初始化、激活进程0、创建进程1、创建进程2的层次划分更清晰。各章内容的分量也比较均衡，阅读感受会更好。

根据读者的反馈意见，第2版增加了一些示意图，在源代码中增加了大量的注释，对操作系统的架构表述得更直观，对源代码讲解得更细致。这些是第2版改动最大、下功夫最多的地方。希望我们的努力能给读者带来更多的帮助。

本书内容及特色

在本书的讲解过程中，我们不仅详细分析了源代码、分析了操作系统的执行序，还特别分析了操作系统都做了哪些“事”，并且对于“事”与“事”之间的关系和来龙去脉，这些“事”意味着什么，为什么要做这些“事”，这些“事”背后的设计思想是什么……都做了非常详细且深入的分析。

更重要的是，对于所有重要的阶段，我们几乎都用图解的方式把操作系统在内存中的实际运行状态精确地表示了出来。我们用600 dpi的分辨率精心绘制了300多张图，图中表现的运行时结构和状态与操作系统实际运行的真实状态完全吻

合。对每一条线、每一个色块、每一个位置、每一个地址及每一个数字，我们都经过了认真反复地推演和求证，并最终在计算机上进行了核对和验证。看了这些绘制精美的图后，读者的头脑中就不再是一行行、一段段枯燥的、令人眩晕的源代码，而是立体呈现的一件件清晰的“事”，以及这些“事”在内存中直截了当、清晰鲜活的画面。用这样的方法讲解操作系统是本书的一大特色。理解这些图要比理解源代码和文字容易得多。毫不夸张地说，只要你能理解这些图，你就理解了操作系统的80%。这时你可以自豪地说，你比大多数用别的方法学过操作系统的人的水平都要高出一大截。

作者和机械工业出版社的编辑做了大量的检索工作。就我们检索的范围而言，这样的创作方法及具有这样特色的操作系统专著在世界范围都是第一次。

本书分三部分来讲解Linux操作系统：第一部分（第1～4章）分析了从开机加电到操作系统启动完成并进入怠速状态的整个过程；第二部分（第5～8章）讲述了操作系统进入系统怠速后，在执行用户程序的过程中，操作系统和用户进程的实际运行过程和状态；第三部分（第9章）阐述整个Linux操作系统的设计指导思想，是从微观到宏观的回归。

第一部分中，我们详细讲解了开机加电启动BIOS，通过BIOS加载操作系统程序，对主机的初

始化，打开保护模式和分页，调用main函数，创建进程0、进程1、进程2以及shell进程，并且具备用文件的形式与外设交互。

第二部分中，我们设计了几个尽可能简单又有代表性的应用程序，并以这些程序的执行为引导，详细讲解了安装文件系统、文件操作、用户进程与内存管理、多个进程对文件的操作以及进程间通信。

我们将操作系统的原理自然而然地融入了讲解真实操作系统的实际运行过程中。在读者看来，操作系统原理不再是空对空的、“文科”概念的计算机理论，而是既有完整且体系的理论，又有真实、具体、实际的代码和案例，理论与实际紧密结合。

第三部分是全书水平最高的部分，详细阐述了主奴机制以及实现主奴机制的三项关键技术：保护和分页、特权级、中断，分析了保障主奴机制实现的决定性因素—先机，还详细讲解了缓冲区、共享页面、信号、管道的设计指导思想。我们尝试从操作系统设计者的视角讲解操作系统的设计指导思想。希望能够帮助读者用体系的思想理解、把握、驾驭整个操作系统以及背后的设计思想和设计意图。

在本书中，我们详细讲解了大家在学习操作系统的过程中可能会遇到的每一个难点，如main函数中的pause（）调用，虽然已经找不到后续代码，但该调用结束后，程序仍然执行的原因是：中断已经打开，进程调度就开始了，而此时可以

调度的进程只有进程1，所以后续的代码应该从进程1处继续执行.....

我们还对读者不容易理解和掌握的操作系统特有的底层代码的一些编程技巧做了详细的讲解，如用模拟call的方法，通过ret指令“调用”main函数.....

总之，我们所做的一切努力就是想真正解决读者遇到的实际问题和难题，给予读者有效的帮助。我们盼望即使是刚刚考入大学的学生也有兴趣和信心把这本书读下去；我们同样希望即使是对操作系统源代码很熟悉的读者，这本书也能给他们一些不同的视角、方法和体系性思考。

为什么本书选用Linux 0.11内核

这本书选用的是Linux 0.11操作系统源代码。对为什么选用Linux 0.11而不是最新版本，赵炯先生有过非常精彩的论述。我们认为赵先生的论述是非常到位的。

我们不妨看一下Linux最新的版本2.6，代码量大约在千万行这个量级，去掉其中的驱动部分，代码量仍在百万行这个量级。一个人一秒钟看一行，一天看8小时，中间不吃、不喝、不休息，也要看上几个月，很难想象如何去理解。

就算我们坚持要选用Linux 2.6，就算我们写上2000页（书足足会有十几厘米厚），所有的篇幅都用来印代码，也只能印上不到十分之一的代

码。所以，即使是这么不切实际的篇幅，也不可能整体讲解Linux 2.6。读者会逐渐明白，对于理解和掌握操作系统而言，真正有价值的是整体、是体系，而不是局部。

Linux 0.11的内核代码虽然只有约两万行，但却是一个实实在在、不折不扣的现代操作系统。因为它具有现代操作系统最重要的特征——支持实时多任务，所以必然支持保护和分页……而且它还是后续版本的真正的始祖，有着内在的、紧密的传承关系。读者更容易看清设计者最初的、最根本的设计意图和设计指导思想。

Linux 0.11已经问世20多年了，被世人广为研究和学习。换一个角度看，要想对众人熟悉的事

物和领域讲出新意和特色，对作者来说也是一个强有力的挑战。

致谢

首先，感谢机械工业出版社华章公司的副总经理温莉芳女士以及其他领导，是他们的决心和决策成就了这本书，并且在几乎所有方面给予了强有力的支持。特别令人感动的是他们主动承担了全部的出版风险，同时给予了作者最好的条件，让我们看到一个大出版社的气度和风范。

其次，特别感谢机械工业出版社华章公司的编辑杨福川。杨先生的鉴赏力和他的事业心以及他对工作认真负责的态度为这本书的出版打开了大门。杨先生对读者的理解以及他的计算机专业素养使得他有能力对这本书给予全方位的指导和

帮助，使我们对这本书整体修改了多次，使之更贴近读者，可读性更好。

还要感谢我们和杨福川共同的朋友张国强先生和杨缙女士。

最后，感谢我们的家人和朋友，是他们坚定的支持才使得整个团队能够拒绝方方面面、形形色色的诱惑，放弃普遍追求的短期利益；我们在常人难以想象的艰苦条件下，长时间专注于操作系统、计算机语言、编译器、计算机体系结构等基础性学科的研究。因为我们认认真真、踏踏实实、不为名利，只为做一点实在、深入的工作，积累了十年的经验，打造了一支敢想、敢干、敢打、敢拼、不惧世界顶级强敌的队伍。这些是本书的基础。

杨力祥

中国科学院研究生院

2013年1月

第1章 从开机加电到执行main函数之前的过程

从开机到main函数的执行分三步完成，目的是实现从启动盘加载操作系统程序，完成执行main函数所需要的准备工作。第一步，启动BIOS，准备实模式下的中断向量表和中断服务程序；第二步，从启动盘加载操作系统程序到内存，加载操作系统程序的工作就是利用第一步中准备的中断服务程序实现的；第三步，为执行32位的main函数做过渡工作。本章将详细分析这三步在计算机中是如何完成的，以及每一步在内存中都做了些什么。

小贴士

实模式（Real Mode）是Intel 80286和之后的80x86兼容CPU的操作模式（应该包括8086）。实模式的特性是一个20位的存储器地址空间（ $2^{20}=1\,048\,576$ ，即1 MB的存储器可被寻址），可以直接软件访问BIOS以及周边硬件，没有硬件支持的分页机制和实时多任务概念。从80286开始，所有的80x86 CPU的开机状态都是实模式；8086等早期的CPU只有一种操作模式，类似于实模式。

1.1 启动BIOS，准备实模式下的中断向量表和中断服务程序

相信大家都知道一台计算机必须要安装一个所谓“操作系统”的软件，才能让我们使用计算

机，否则计算机将是一堆毫无生命力的冰冷的硬家伙。在为计算机安装了操作系统后，当你按下计算机电源按钮的那一刻，计算机机箱传来了嗡嗡的声音。这时你感觉到，计算机开始启动工作了。然而，在计算机的启动过程中，操作系统底层与计算机硬件之间究竟做了哪些复杂的交互动作？下面我们将根据操作系统实际的启动和运行过程对此进行逐步的剖析和讲解。

计算机的运行是离不开程序的。然而，加电的一瞬间，计算机的内存中，准确地说是**RAM**中，空空如也，什么程序也没有。软盘里虽然有操作系统程序，但**CPU**的逻辑电路被设计为只能运行内存中的程序，没有能力直接从软盘运行操作系统^[1]。如果要运行软盘中的操作系统，必须

将软盘中的操作系统程序加载到内存（RAM）中。

特别注意

我们假定本书所用的计算机是基于IA—32系列CPU，安装了标准单色显示器、标准键盘、一个软驱、一块硬盘、16 MB内存，在内存中开辟了2 MB内存作为虚拟盘，并在BIOS中设置软驱为启动设备。后续所有的讲解都以此为基础。

小贴士

RAM（Random Access Memory）：随机存取存储器，常见的内存条就是一类RAM，其特点是加电状态下可任意读、写，断电后信息消失。

问题：在RAM中什么程序也没有的时候，谁来完成加载软盘中操作系统的任务呢？

答案是：BIOS。

1.1.1 BIOS的启动原理

在了解BIOS是如何将操作系统程序加载到内存中之前，我们先来了解一下BIOS程序自身是如何启动的。从我们使用计算机的经验得知：要想执行一个程序，必须在窗口中双击它，或者在命令行界面中输入相应的执行命令。从计算机底层机制上讲，其实是在一个已经运行起来的操作系统的可视化界面或命令行界面中执行一个程序。但是，在开机加电的一瞬间，内存中什么程序也没有，没有任何程序在运行，不可能有操作系

统，更不可能有操作系统的用户界面。我们无法人为地执行BIOS程序，那么BIOS程序又是由谁来执行的呢？

秘诀是：0xFFFF0！

从体系的角度看，不难得出这样的结论：既然用软件方法不可能执行BIOS，就只能靠硬件方法完成了。

从硬件角度看，Intel 80x86系列的CPU可以分别在16位实模式和32位保护模式下运行。为了兼容，也为了解决最开始的启动问题，Intel将所有80x86系列的CPU，包括最新型号的CPU的硬件都设计为加电即进入16位实模式状态运行。同时，还有一点非常关键的是，将CPU硬件逻辑设计为

加电瞬间强行将CS的值置为0xF000、IP的值置为0xFFFF0，这样CS: IP就指向0xFFFF0这个地址位置，如图 [2] 1-1所示。从图1-1中可以清楚地看到，0xFFFF0指向了BIOS的地址范围。

小贴士

IP/EIP（Instruction Pointer）： 指令指针寄存器，存在于CPU中，记录将要执行的指令在代码段内的偏移地址，和CS组合即为将要执行的指令的内存地址。实模式为绝对地址，指令指针为16位，即IP；保护模式下为线性地址，指令指针为32位，即EIP。



图 1-1 启动时BIOS在内存的状态及初始执行位置

小贴士

CS (Code Segment Register)：代码段寄存器，存在于CPU中，指向CPU当前执行代码在内存中的区域（定义了存放代码的存储器的起始地址）。

注意，这是一个纯硬件完成的动作！如果此时这个位置没有可执行代码，那么就什么也不用说了，计算机就此死机。反之，如果这个位置有可执行代码，计算机将从这里的代码开始，沿着后续程序一直执行下去。

BIOS程序的入口地址恰恰就是0xFFFF0！也就是说，BIOS程序的第一条指令就设计在这个位置。

[1] Linus写Linux0.11是在1991年年底。那时，很多计算机是从软盘启动的。他为Linux0.11设计的系统启动盘是软盘。

[2] 本书中的大部分图都是依照计算机实际运行时的内存真实状态，严格按比例以600 dpi分辨率精确绘制的，所以有些内存区域因为体积比较小，

在图中占的位置也比较小。请大家阅读时仔细辨认。

1.1.2 BIOS在内存中加载中断向量表和中 断服务程序

BIOS程序的代码量并不大，却非常精深，需要对整个计算机硬件体系结构非常熟悉才能看得明白。要想把BIOS是如何运行的讲清楚，也得写很厚的一本书，这显然超出了本书的主题和范围。我们的主题是操作系统，所以只把与启动操作系统有直接关系的部分简单地讲解一下。

BIOS程序被固化在计算机主机板上的一块很小的ROM芯片里。通常不同的主机板所用的BIOS也有所不同。就启动部分而言，各种类型的BIOS的基本原理大致相似。为了便于大家理解，我们选用的BIOS程序只有8 KB，所占地址段为

0xFE000~0xFFFFF，如图1-1所示。现在CS: IP已经指向0xFFFF0这个位置了，这意味着BIOS开始启动了。随着BIOS程序的执行，屏幕上会显示显卡的信息、内存的信息.....说明BIOS程序在检测显卡、内存.....这期间，有一项对启动

（boot）操作系统至关重要的工作，那就是BIOS在内存中建立中断向量表和中断服务程序。

小贴士

ROM（Read Only Memory）：只读存储器。现在通常用闪存芯片做ROM。虽然闪存芯片在特定的条件下是可写的，但在谈到主机板上存储BIOS的闪存芯片时，业内人士把它看做ROM。ROM有一个特性，就是断电之后仍能保存信息，这一点和硬盘类似。

BIOS程序在内存最开始的位置（0x00000）用1 KB的内存空间（0x00000～0x003FF）构建中断向量表，在紧挨着它的位置用256字节的内存空间构建BIOS数据区（0x00400～0x004FF），并在大约57 KB以后的位置（0x0E05B）加载了8 KB左右的与中断向量表相应的若干中断服务程序。图1-2中精确地标注了这些位置。

小贴士

一个容易计算的方法：0x00100是256字节，0x00400就是4×256字节=1024字节，也就是1 KB。因为是从0x00000开始计算，所以1 KB的高地址端不是0x00400，而是0x00400-1，也就是0x003FF。

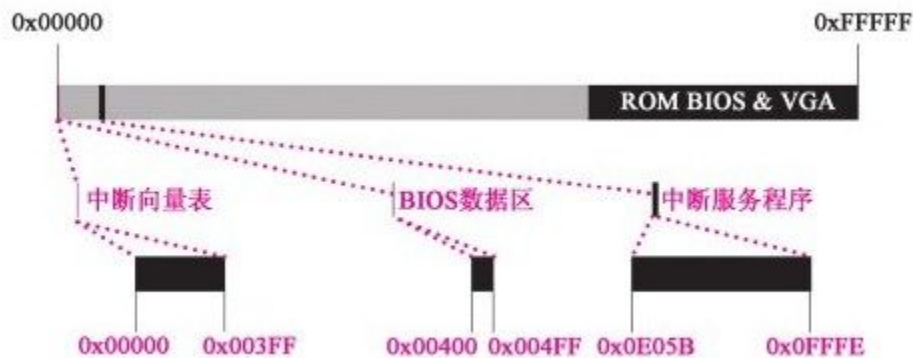


图 1-2 BIOS在内存中加载中断向量表和中断服务程序

中断向量表中有256个中断向量，每个中断向量占4字节，其中两个字节是CS的值，两个字节是IP的值。每个中断向量都指向一个具体的中断服务程序。

下面将详细讲解后续程序是如何利用这些中断服务程序把系统内核程序从软盘加载至内存的。

小贴士

INT (INTerrupt)：中断，顾名思义，中途打断一件正在进行中的事。其最初的意思是：外在的事件打断正在执行的程序，转而执行处理这个事件的特定程序，处理结束后，回到被打断的程序继续执行。现在，可以先将中断理解为一种技术手段，在这一点上与C语言的函数调用有些类似。

中断对操作系统来说是一个意义重大的概念，后面我们还会深入讨论。

1.2 加载操作系统内核程序并为保护模式做准备

从现在开始，就要执行真正的boot操作了，即把软盘中的操作系统程序加载至内存。对于Linux 0.11操作系统而言，计算机将分三批逐次加载操作系统的内核代码。第一批由BIOS中断int 0x19把第一扇区bootsect的内容加载到内存；第二批、第三批在bootsect的指挥下，分别把其后的4个扇区和随后的240个扇区的内容加载至内存。

1.2.1 加载第一部分内核代码——引导程序（bootsect）

按照我们使用计算机的经验，如果在开机的时候马上按**Del**键，屏幕上会显示一个**BIOS**画面，可以在里面设置启动设备。现在我们基本上都是将硬盘设置为启动盘。Linux 0.11是1991年设计的操作系统，那时常用的启动设备是软驱以及其中的软盘。站在体系结构的角度看，从软盘启动和从硬盘启动的基本原理和机制是类似的。

经过执行一系列**BIOS**代码之后，计算机完成了自检等操作（这些和我们讲的启动操作系统没有直接的关系，读者不必关心）。由于我们把软盘设置为启动设备，计算机硬件体系结构的设计与**BIOS**联手操作，会让CPU接收到一个int 0x19中断。CPU接收到这个中断后，会立即在中断向量表中找到int 0x19中断向量。我们在图1-3的左下

方可以看到int 0x19中断向量在内存中所在的准确位置，这个位置几乎紧挨着内存的0x00000位置。

接下来，中断向量把CPU指向0x0E6F2，这个位置就是int 0x19相对应的中断服务程序的入口地址，即图1-3所示的“启动加载服务程序”的入口地址。这个中断服务程序的作用就是把软盘第一扇区中的程序（512 B）加载到内存中的指定位置。这个中断服务程序的功能是BIOS事先设计好的，代码是固定的，与Linux操作系统无关。无论Linux 0.11的内核是如何设计的，这段BIOS程序所要做的就是“找到软盘”并“加载第一扇区”，其余的它什么都不知道，也不必知道。

小贴士

中断向量表（Interrupt Vector Table）：实模式中中断机制的重要组成部分，记录所有中断号对应的中断服务程序的内存地址。

中断服务（Interrupt Service）程序：通过中断向量表的索引对中断进行响应服务，是一些具有特定功能的程序。

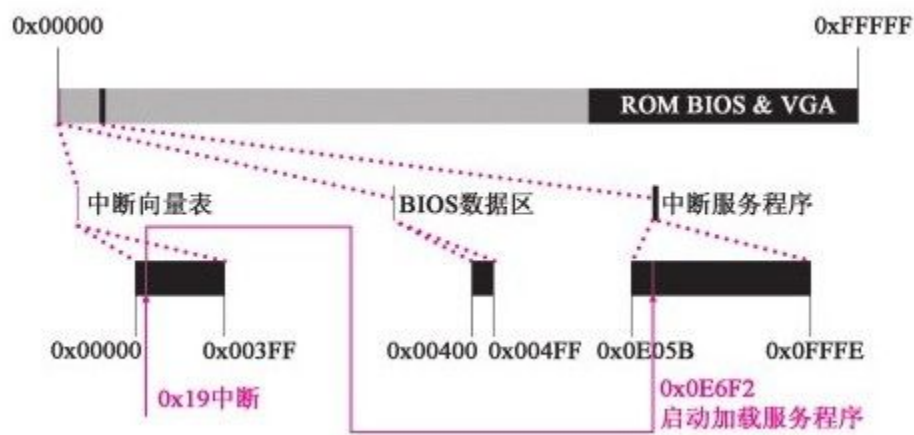


图 1-3 响应int 0x19中断

按照这个简单、“生硬”的规则，int 0x19中断向量所指向的中断服务程序，即启动加载服务程

序，将软驱0号磁头对应盘面的0磁道1扇区的内容复制至内存0x07C00处。我们可以在图1-4的左边看到第一扇区加载的具体位置。

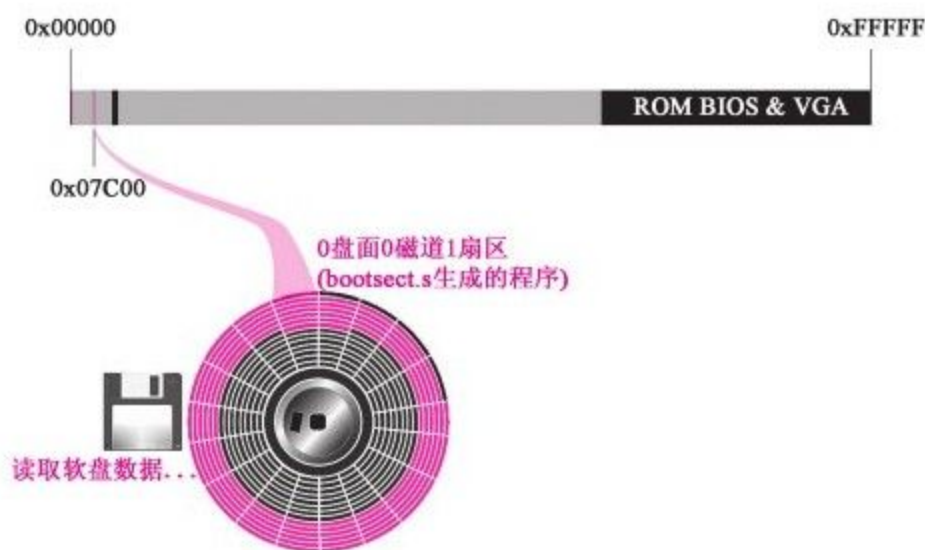


图 1-4 把软盘第一扇区中的程序加载到内存中的指定位置

这个扇区里的内容就是Linux 0.11的引导程序，也就是我们将要讲解的bootsect，其作用就是陆续把软盘中的操作系统程序载入内存。这样制

作的第一扇区就称为启动扇区（boot sector）。第一扇区程序的载入，标志着Linux 0.11中的代码即将发挥作用了。

这是非常关键的动作，从此计算机开始和软盘上的操作系统程序产生关联。第一扇区中的程序由bootsect.s中的汇编程序汇编而成（以后简称bootsect）。这是计算机自开机以来，内存中第一次有了Linux操作系统自己的代码，虽然只是启动代码。

至此，已经把第一批代码bootsect从软盘载入计算机的内存了。下面的工作就是执行bootsect把软盘的第二批、第三批代码载入内存。

点评

注意：BIOS程序固化在主机板上的ROM中，是根据具体的主机板而不是根据具体的操作系统设计的。

理论上，计算机可以安装任何适合其安装的操作系统，既可以安装Windows，也可以安装Linux。不难想象每个操作系统的设计者都可以设计出一套自己的操作系统启动方案，而操作系统和BIOS通常是由不同的专业团队设计和开发的，为了能协同工作，必须建立操作系统和BIOS之间的协调机制。

与已有的操作系统建立一一对应的协调机制虽然麻烦，但尚有可能，难点在于与未来的操作系统应该如何建立协调机制。现行的方法是“两头约定”和“定位识别”。

对操作系统（这里指Linux 0.11）而言，“约定”操作系统的设计者必须把最开始执行的程序“定位”在启动扇区（软盘中的0盘面0磁道1扇区），其余的程序可以依照操作系统的设计顺序加载在后续的扇区中。

对BIOS而言，“约定”接到启动操作系统的命令，“定位识别”只从启动扇区把代码加载到0x07C00（BOOTSEG）这个位置（参见Seabios 0.6.0/Boot.c文件中的boot_disk函数）。至于这个扇区中是否是启动程序、是什么操作系统，则不闻不问、一视同仁。如果不是启动代码，只会提示错误，其余是用户的责任，与BIOS无关。

这样构建协调机制的好处是站在整个体系的高度，统一设计、统一安排，简单、有效。只要

BIOS和操作系统的生产厂商开发的所有系统版本全部遵循此机制的约定，就可以各自灵活地设计出具有自己特色的系统版本。

1.2.2 加载第二部分内核代码——setup

1.bootsect对内存的规划

BIOS已经把bootsect也就是引导程序载入内存了，现在它的作用就是把第二批和第三批程序陆续加载到内存中。为了把第二批和第三批程序加载到内存中的适当位置，bootsect首先做的工作就是规划内存。

通常，我们是用高级语言编写应用程序的，这些程序是在操作系统的平台上运行的。我们只管写高级语言的代码、数据。至于这些代码、数据在运行的时候放在内存的什么地方，是否会相互覆盖，我们都不用操心，因为操作系统和高级语言的编译器替我们做了大量的看护工作，确保

不会出错。现在我们讨论的是，操作系统本身使用的是汇编语言，没有高级语言编译器替操作系统提供保障，只有靠操作系统的设计者把内存的安排想清楚，确保无论操作系统如何运行，都不会出现代码与代码、数据与数据、代码与数据之间相互覆盖的情况。为了更准确地理解操作系统的运行机制，我们必须清楚操作系统的设计者是如何规划内存的。

在实模式状态下，寻址的最大范围是1 MB。为了规划内存，bootsect首先设计了如下代码：

```
//代码路径: boot/bootsect.s
```

```
.....
```

```
.globl begtext,begdata,begbss,endtext,enddata,endbss
```

```
.text
```


begtext:

.data

begdata:

.bss

begbss:

.text

SETUPLN=4 ! nr of setup-sectors

BOOTSEG=0x07c0 ! original address of boot-sector

INITSEG=0x9000 ! we move boot here-out of the way

SETUPSEG=0x9020 ! setup starts here

SYSSEG=0x1000 ! system loaded at 0x10000 (65536) .

ENDSEG=SYSSEG+SYSSIZE ! where to stop loading

! ROOT_DEV: 0x000-same type of floppy as boot.

! 0x301-first partition on first drive etc

ROOT_DEV=0x306

.....

这些源代码的作用就是对后续操作所涉及的内存位置进行设置，包括将要加载的setup程序的扇区数（SETUPLen）以及被加载到的位置（SETUPSEG）；启动扇区被BIOS加载的位置（BOOTSEG）及将要移动到的新位置（INITSEG）；内核（kernel）被加载的位置（SYSSEG）、内核的末尾位置（ENDSEG）及根文件系统设备号（ROOT_DEV）。这些位置在图1-5中都被明确地标注了出来。设置这些位置就是为了确保将要载入内存的代码与已经载入内存的代码及数据各在其位，互不覆盖，并且各自有够用的内存空间。大家在后续的章节会逐渐看到内存规划的意义和作用。



图 1-5 实模式下的内存使用规划

从现在起，我们的头脑中要时刻牢记这样一个概念：操作系统的设计者是要全面地、整体地考虑内存的规划的。我们会在后续的章节中不断地了解到，精心安排内存是操作系统设计者时时刻刻都要关心的事。我们带着这样的观念继续了解bootsect程序的执行。

2.复制bootsect

接下来，bootsect启动程序将它自身（全部的512 B内容）从内存0x07C00（BOOTSEG）处复制至内存0x90000（INITSEG）处。这个动作和目标位置如图1-6所示。

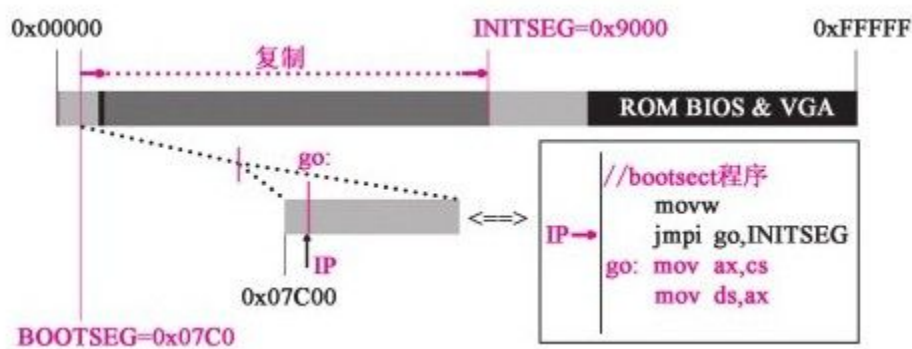


图 1-6 bootsect复制自身

执行这个操作的代码（boot/bootsect.s）如下：

```
//代码路径: boot/bootsect.s
```

```
.....
```

```
entry start
```

```
start:

mov ax, #BOOTSEG

mov ds,ax

mov ax, #INITSEG

mov es,ax

mov cx, #256

sub si,si

sub di,di

rep

movw

.....
```

在这次复制过程中，ds（0x07C0）和si（0x0000）联合使用，构成了源地址0x07C00；es（0x9000）和di（0x0000）联合使用，构成了目的地址0x90000（见图1-6），而mov cx, #256这

一行循环控制量，提供了需要复制的“字”数（一个字为2字节，256个字正好是512字节，也就是第一扇区的字节数）。

通过代码我们还可以看出，图1-5提到的BOOTSEG和INITSEG现在开始发挥作用了。注意，此时CPU的段寄存器（CS）指向0x07C0（BOOTSEG），即原来bootsect程序所在的位置。

点评

由于“两头约定”和“定位识别”，所以在开始时bootsect“被迫”加载到0x07C00位置。现在将自身移至0x90000处，说明操作系统开始根据自己的需要安排内存了。

bootsect复制到新位置完毕后，会执行下面的代码：

```
//代码路径: boot/bootsect.s
```

```
.....
```

```
rep
```

```
movw
```

```
jmp go,INITSEG
```

```
go: mov ax,cs
```

```
mov ds,ax
```

```
.....
```

从图1-6中我们已经了解到当时CS的值为0x07C0，执行完这个跳转后，CS值变为0x9000（INITSEG），IP的值为从0x9000（INITSEG）到**go: mov ax,cs**这一行对应指令的偏移。换句话

说，此时CS: IP指向go: mov ax,cs这一行，程序从这一行开始往下执行。图1-7形象地表示了跳转到go: mov ax,cs这一行执行时CS和IP的状态，如图右下方所示。

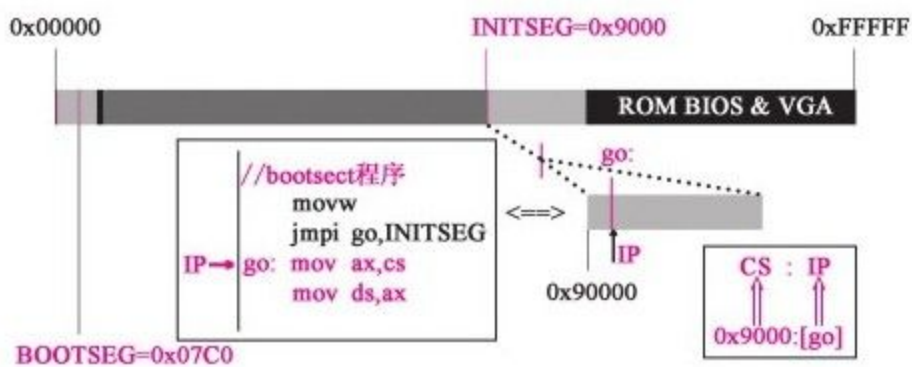


图 1-7 跳转到go处继续执行

此前的0x07C00这个位置是根据“两头约定”和“定位识别”而确定的。从现在起，操作系统已经不需要完全依赖BIOS，可以按照自己的意志把自己的代码安排在内存中自己想要的位置。

点评

```
jmpb go,INITSEG
```

```
go: mov ax,cs
```

这两行代码写得很巧。复制bootsect完成后，在内存的0x07C00和0x90000位置有两段完全相同的代码。请大家注意，复制代码这件事本身也是要靠指令执行的，执行指令的过程就是CS和IP不断变化的过程。执行到`jmpb go,INITSEG`这行之前，代码的作用就是复制代码自身；执行了`jmpb go,INITSEG`之后，程序就转到执行0x90000这边的代码了。Linus的设计意图是想跳转之后，在新位置接着执行后面的`mov ax,cs`，而不是死循环。`jmpb go,INITSEG`与`go: mov ax,cs`配合，巧妙地实

现了“到新位置后接着原来的执行序继续执行下去”的目的。

bootsect复制到了新的地方，并且要在新的地方继续执行。因为代码的整体位置发生了变化，所以代码中的各个段也会发生变化。前面已经改变了CS，现在对DS、ES、SS和SP进行调整。我们看看下面的代码：

```
//代码路径: boot/bootsect.s
```

```
.....
```

```
go:  mov ax,cs
```

```
mov ds,ax
```

```
mov es,ax
```

```
! put stack at 0x9ff00.
```

```
mov ss,ax
```

```
mov sp, #0xFF00! arbitrary value > > 512
```

! load the setup-sectors directly after the bootblock.

! Note that'es'is already set up.

.....

上述代码的作用是通过ax，用CS的值0x9000来把数据段寄存器（DS）、附加段寄存器（ES）、栈基址寄存器（SS）设置成与代码段寄存器（CS）相同的位置，并将栈顶指针SP指向偏移地址为0xFF00处。图1-8对此做了非常直观的描述。

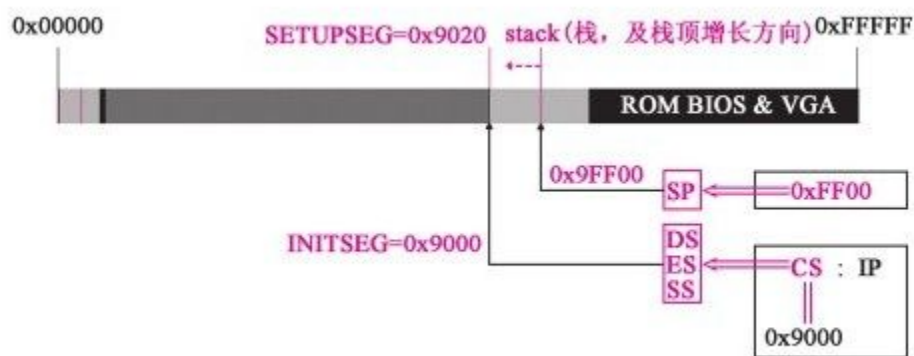


图 1-8 调整各个段寄存器值

下面着重介绍一下与栈操作相关的寄存器的设置。SS和SP联合使用，就构成了栈数据在内存中的位置值。对这两个寄存器的设置为后面程序的栈操作（如push、pop等）打下了基础。

现在可以观察一下bootsect中的程序，在执行设置SS和SP的代码之前，没有出现过栈操作指令，而在此之后就陆续使用。这里对SS和SP进行的设置是分水岭。它标志着从现在开始，程序可以执行一些更为复杂的数据运算类指令了。

栈操作是有方向的。图1-8中标识了压栈的方向，注意是由高地址到低地址的方向。

小贴士

DS/ES/FS/GS/SS：这些段寄存器存在于CPU中，其中SS（Stack Segment）指向栈段，此区域将按栈机制进行管理。

SP（Stack Pointer）：栈顶指针寄存器，指向栈段的当前栈顶。

注意：很多计算机书上使用“堆栈”这个词。本书用堆、栈表示两个概念。栈表示stack，特指在C语言程序的运行时结构中，以“后进先出”机制运作的内存空间；堆表示heap，特指用C语言库函数malloc创建、free释放的动态内存空间。

至此，bootsect的第一步操作，即规划内存并把自身从0x07C00的位置复制到0x90000的位置的动作已经完成了。

3.将setup程序加载到内存中

下面，bootsect程序要执行它的第二步工作：
将setup程序加载到内存中。

加载setup这个程序，要借助BIOS提供的int 0x13中断向量所指向的中断服务程序（也就是磁盘服务程序）来完成。图1-9标注了int 0x13中断向量的位置以及这个中断向量所指向的磁盘服务程序的入口位置。

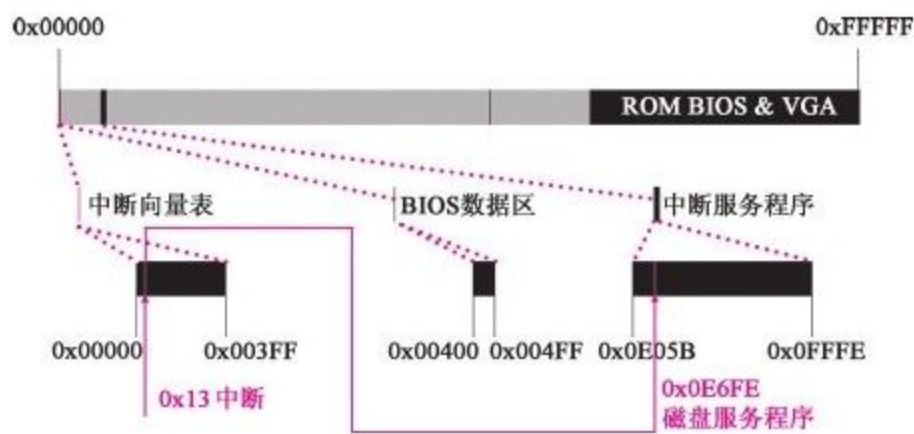


图 1-9 调用int 0x13中断

这个中断服务程序的执行过程与图1-3和图1-4中讲解过的int 0x19中断向量所指向的启动加载服务程序不同。

int 0x19中断向量所指向的启动加载服务程序是BIOS执行的，而int 0x13的中断服务程序是Linux操作系统自身的启动代码bootsect执行的。

int 0x19的中断服务程序只负责把软盘的第一扇区的代码加载到0x07C00位置，而int 0x13的中断服务程序则不然，它可以根据设计者的意图，把指定扇区的代码加载到内存的指定位置。

针对服务程序的这个特性，使用int 0x13中断时，就要事先将指定的扇区、加载的内存位置等信息传递给服务程序，即传参。执行代码如下：

//代码路径: boot/bootsect.s

.....! 注意: SETUPLEN为4

load_setup:

mov dx, #0x0000! drive 0, head 0

mov cx, #0x0002! sector 2, track 0

mov bx, #0x0200! address=512, in INITSEG

mov ax, #0x0200+SETUPLEN! service 2, nr of sectors

int 0x13! read it

jnc ok_load_setup! ok-continue

mov dx, #0x0000

mov ax, #0x0000! reset the diskette

int 0x13

j load_setup

.....

从代码开始处的4个mov指令可以看出, 系统给BIOS中断服务程序传参是通过几个通用寄存器

实现的。这是汇编程序的常用方法，与C语言的函数调用形式有很大不同。

参数传递完毕后，执行`int 0x13`指令，产生`0x13`中断，通过中断向量表找到这个中断服务程序，将软盘第二扇区开始的4个扇区，即`setup.s`对应的程序加载至内存的`SETUPSEG`（`0x90200`）处。根据对图1-5的讲解，复制后的`bootsect`的起始位置是`0x90000`，占用512字节的内存空间。不难看出`0x90200`紧挨着`bootsect`的尾端，所以`bootsect`和`setup`是连在一起的。图1-10表示了软盘中所要加载的扇区位置和扇区数，以及载入内存的目标位置和占用空间。

现在，操作系统已经从软盘中加载了5个扇区的代码。等`bootsect`执行完毕后，`setup`这个程序就

要开始工作了。

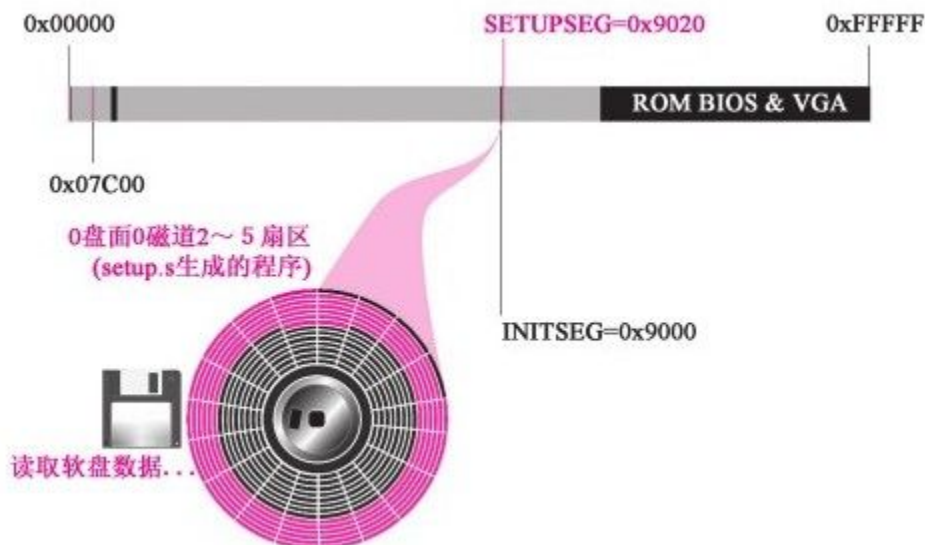


图 1-10 加载setup程序

注意，图1-8中SS: SP指向的位置为0x9FF00，这与setup程序的起始位置0x90200还有很大的距离，即便setup加载进来后，系统仍然有足够的内存空间用来执行数据压栈操作；而且，在启动部分，要压栈的数据毕竟也是有限的。大

家在后续的章节中会逐渐体会到，设计者在此是进行过精密测算的。

1.2.3 加载第三部分内核代码——system 模块

第二批代码已经载入内存，现在要加载第三批代码。仍然使用BIOS提供的int 0x13中断，如图1-11所示，方法与图1-9所示的方法基本相同。

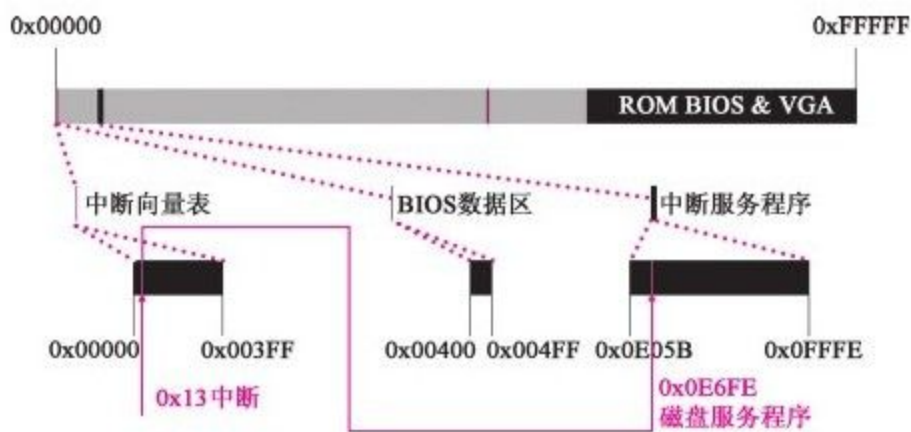


图 1-11 再次调用int 0x13中断

接下来，bootsect程序要执行第三批程序的载入工作，即将系统模块载入内存。

这次载入从底层技术上看，与前面的setup程序的载入没有本质的区别。比较突出的特点是这次加载的扇区数是240个，足足是之前的4个扇区的60倍，所需时间也是几十倍。为了防止加载期间用户误认为是机器故障而执行不适当的操作，Linux在此设计了显示一行屏幕信息“Loading system.....”以提示用户计算机此时正在加载系统。值得注意的是，此时操作系统的主函数还没有开始执行，在屏幕上显示一行字符串远没有用C语言写一句printf (“Loading system.....\n”)调用那么容易，所有工作都要靠一行一行的汇编代码来实现。从体系结构的角度看，显示器也是一个外设，所以还要用到其他BIOS中断。这些代码比较多，对理解操作系统的启动原理没有特别直接的帮助，只要知道大意就可以了。我们真正

需要掌握的是，bootsect借着BIOS中断int 0x13，将240个扇区的system模块加载进内存。加载工作主要是由bootsect调用read_it子程序完成的。这个子程序将软盘第六扇区开始的约240个扇区的system模块加载至内存的SYSSEG（0x10000）处往后的120 KB空间中。图1-12对system模块所占用的内存空间给出了形象的说明。

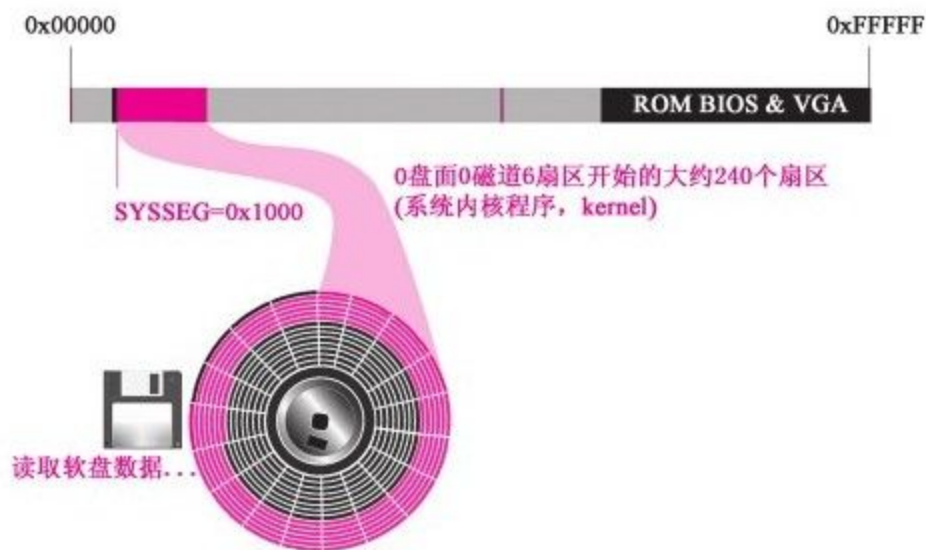


图 1-12 加载system模块

由于是长时间操作软盘，所以需要软盘设备进行更多的监控，对读盘结果不断地进行检测。因此read_it后续的调用步骤比较多一些。但读盘工作最终是由0x13对应的中断服务程序完成的。

到此为止，第三批程序已经加载完毕，整个操作系统的代码已全部加载至内存。bootsect的主体工作已经做完了，还有一点小事，就是要再次确定一下根设备号，如图1-13所示。

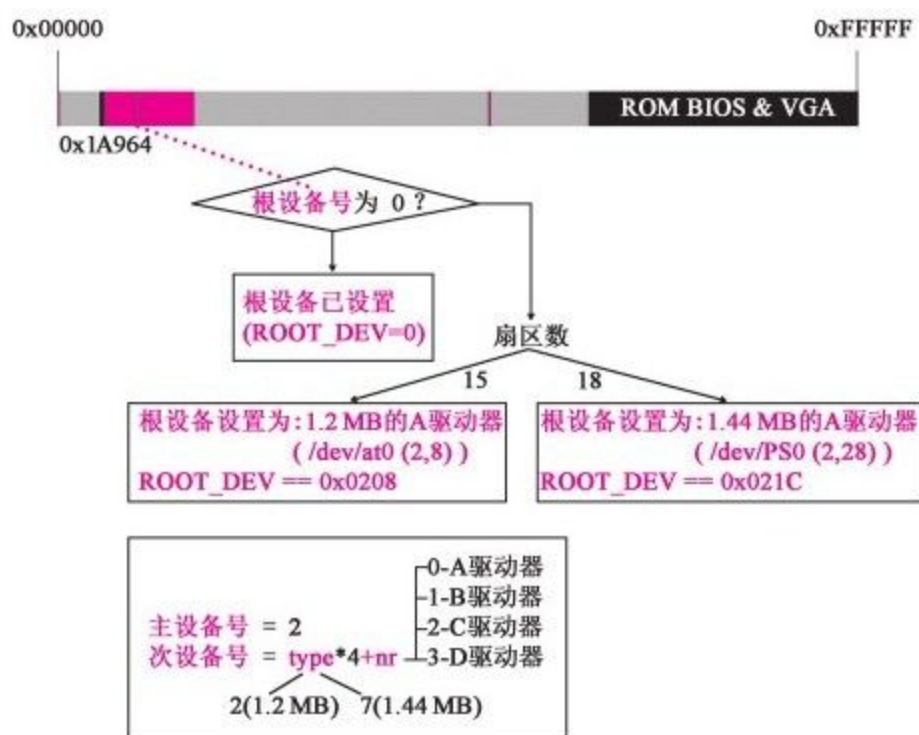


图 1-13 确认根设备号

小贴士

根文件系统设备（Root Device）：Linux 0.11 使用Minix操作系统的文件系统管理方式，要求系统必须存在一个根文件系统，其他文件系统挂接其上，而不是同等地位。Linux 0.11没有提供在设

备上建立文件系统的工具，故必须在一个正在运行的系统上利用工具（类似FDISK和Format）做出一个文件系统并加载至本机。因此Linux 0.11的启动需要两部分数据，即系统内核镜像和根文件系统。

注意：这里的文件系统指的不是操作系统内核中的文件系统代码，而是有配套的文件系统格式的设备，如一张格式化好的软盘。

因为本书假设所用的计算机安装了一个软盘驱动器、一个硬盘驱动器，在内存中开辟了2 MB的空间作为虚拟盘（见第2章的main函数），并在BIOS中设置软盘驱动器为启动盘，所以，经过一系列检测，确认计算机中实际安装的软盘驱动器为根设备，并将信息写入机器系统数据。第2章中

main函数一开始就用机器系统数据中的这个信息设置根设备，并为“根文件系统加载”奠定基础。

执行代码如下：

```
//代码路径: boot/bootsect.s

.....

seg cs

mov ax,root_dev

cmp ax, #0

jne root_defined

seg cs

mov bx,sectors

mov ax, #0x0208! /dev/ps0-1.2Mb

cmp bx, #15

je root_defined

mov ax, #0x021c! /dev/PS0-1.44Mb
```

```
cmp bx, #18
```

```
je root_defined
```

```
undef _root:
```

```
jmp undef_root
```

`root_defined:` ! 根据前面检测计算机中实际安装的驱动器信息, 确认根设备

```
seg cs
```

```
mov root_dev,ax
```

```
.....
```

`.org 508`! 注意: 508即为0x1FC, 当前段是0x9000, 所以地址是0x901FC

```
root_dev:
```

```
.word ROOT_DEV
```

```
boot_flag:
```

```
.word 0xAA55
```

```
.....
```

现在, bootsect程序的任务都已经完成!

下面要通过执行“`jmp 0, SETUPSEG`”这行语句跳转至0x90200处，就是前面讲过的第二批程序——`setup`程序加载的位置。CS: IP指向`setup`程序的第一条指令，意味着由`setup`程序接着`bootsect`程序继续执行。图1-14形象地描述了跳转到`setup`程序后的起始状态，对应的代码如下：

```
//代码路径: boot/bootsect.s
```

```
.....
```

```
jmp 0, SETUPSEG
```

```
.....
```

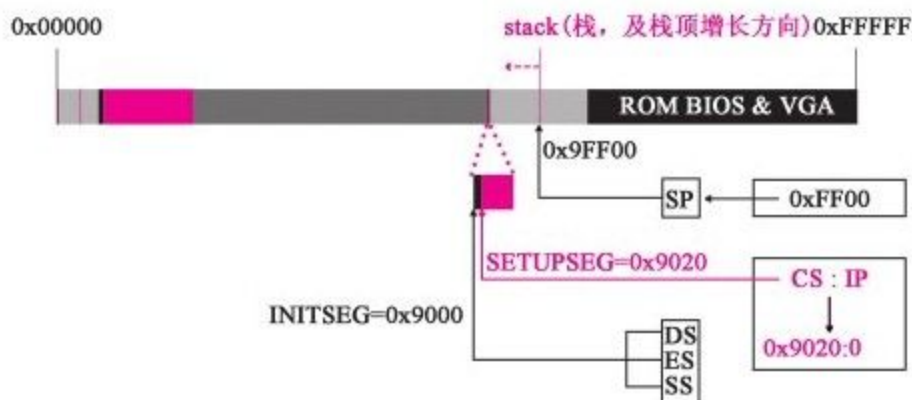


图 1-14 setup开始执行

setup程序现在开始执行。它做的第一件事情就是利用BIOS提供的中断服务程序从设备上提取内核运行所需的机器系统数据，其中包括光标位置、显示页面等数据，并分别从中断向量0x41和0x46向量值所指的内存地址处获取硬盘参数表1、硬盘参数表2，把它们存放在0x9000: 0x0080和0x9000: 0x0090处。

这些机器系统数据被加载到内存的0x90000～0x901FC位置。图1-15标出了其内容及准确的位置。这些数据将在以后main函数执行时发挥重要作用。

提取机器系统数据的具体代码如下：

//代码路径: boot/setup.s

mov ax, #INITSEG! this is done in bootsect already,but.....

mov ds,ax

mov ah, #0x03! read cursor pos

xor bh,bh

int 0x10! save it in known place,con_init fetches

mov[0], dx! it from 0x90000.

! Get memory size (extended mem,kB)

mov ah, #0x88

int 0x15

mov[2], ax

mov cx, #0x10

mov ax, #0x00

rep

stosb

.....

这段代码大约70行，由于篇幅限制，我们省略了大部分代码。

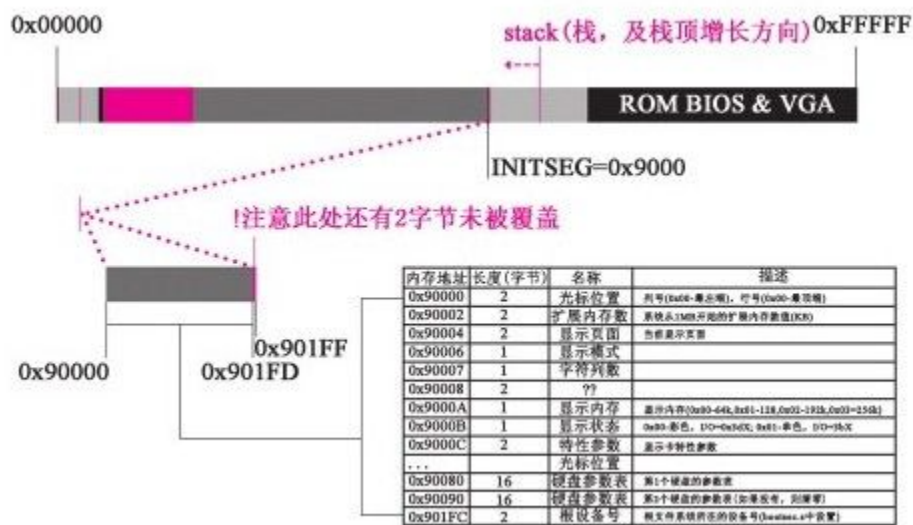


图 1-15 加载机器系统数据

注意，BIOS提取的机器系统数据将覆盖bootsect程序所在部分区域。这些数据由于是要留用的，所以在它们失去使用价值之前，一定不能被覆盖掉。

点评

机器系统数据所占的内存空间为0x90000～0x901FD，共510字节，即原来bootsect只有2字节未被覆盖。可见，操作系统对内存的使用是非常严谨的。在空间上，操作系统对内存严格按需使用，要加载的数据刚好占用一个扇区的位置（只差2字节），而启动扇区bootsect又恰好是一个扇区，内存的使用规划像一个账本，前后对应；在时间上，使用完毕的空间立即挪作他用，启动扇区bootsect程序刚结束其使命，执行setup时立刻就将其用数据覆盖，内存的使用率极高。虽然这与当时的硬件条件有限不无关系，但这种严谨的内存规划风格是很值得学习的。

到此为止，操作系统内核程序的加载工作已经完成。接下来的操作对Linux 0.11而言具有战略

意义。系统通过已经加载到内存中的代码，将实现从实模式到保护模式的转变，使Linux 0.11真正成为“现代”操作系统。

1.3 开始向32位模式转变，为main函数的调用做准备

接下来，操作系统要使计算机在32位保护模式下工作。这期间要做大量的重建工作，并且持续工作到操作系统的main函数的执行过程中。在本节中，操作系统执行的操作包括打开32位的寻址空间、打开保护模式、建立保护模式下的中断响应机制等与保护模式配套的相关工作、建立内存的分页机制，最后做好调用main函数的准备。

1.3.1 关中断并将system移动到内存地址起始位置0x00000

如图1-16所示，这个准备工作先要关闭中断，即将CPU的标志寄存器（EFLAGS）中的中断允许标志（IF）置0。这意味着，程序在接下来的执行过程中，无论是否发生中断，系统都不再对此中断进行响应，直到下一章要讲解的main函数中能够适应保护模式的中断服务体系被重建完毕才会打开中断，而那时候响应中断的服务程序将不再是BIOS提供的中断服务程序，取而代之的是由系统自身提供的中断服务程序。代码如下：

```
//代码路径: boot/setup.s

.....

cli ! no interrupts allowed !

.....
```

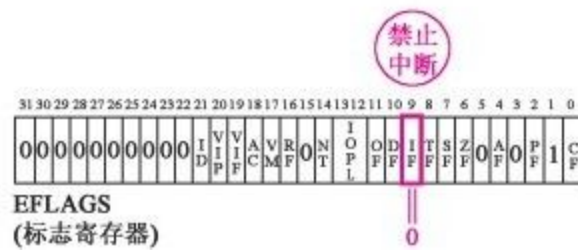


图 1-16 关中断

小贴士

EFLAGS: 标志寄存器，存在于CPU中，32位，包含一组状态标志、控制标志及系统标志。如第0位的CF（Carry Flag）为CPU计算用到的进位标志，及图1-16所示的关中断操作涉及的第9位IF（Interrupt Flag）中断允许标志。

点评

关中断（cli）和开中断（sti）操作将在操作系统代码中频繁出现，其意义深刻。慢慢的你会

发现，cli、sti总是在一个完整操作过程的两头出现，目的是避免中断在此期间的介入。接下来的代码将为操作系统进入保护模式做准备。此处即将进行实模式下中断向量表和保护模式下中断描述符表（IDT）的交接工作。试想，如果没有cli，又恰好发生中断，如用户不小心碰了一下键盘，中断就要切进来，就不得不面对实模式的中断机制已经废除、保护模式的中断机制尚未完成的尴尬局面，结果就是系统崩溃。cli、sti保证了这个过程中，IDT能够完整创建，以避免不可预料中断的进入造成IDT创建不完整或新老中断机制混用。甚至可以理解为cli、sti是为了保护一个新的计算机生命的完整而创建的。

下面， setup程序做了一个影响深远的动作：
将位于0x10000的内核程序复制至内存地址起始位置0x00000处！ 代码如下：

```
//代码路径: boot/setup.s
```

```
.....
```

```
do _move:
```

```
mov es,ax ! destination segment
```

```
add ax, #0x1000
```

```
cmp ax, #0x9000
```

```
jz end_move
```

```
mov ds,ax ! source segment
```

```
sub di,di
```

```
sub si,si
```

```
mov cx, #0x8000
```

```
rep
```

```
movsw
```

```
jmp do_move
```

```
.....
```

图1-17准确标识了复制操作系统内核代码的源位置和目标位置及复制动作的方向。

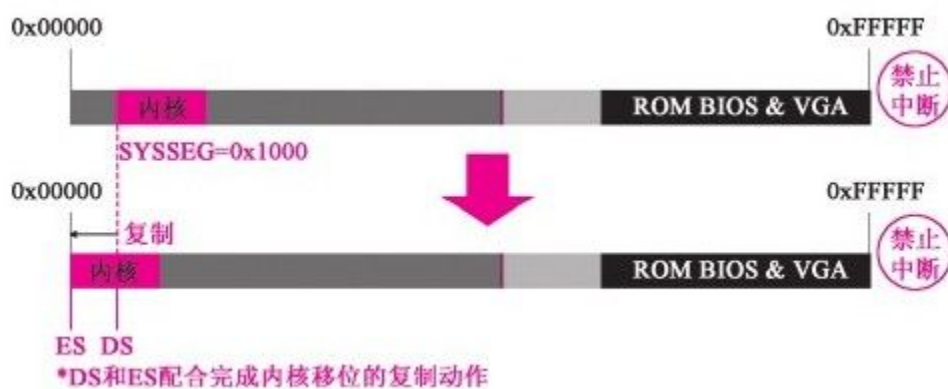


图 1-17 复制system模块至内存起始处

回顾一下图1-2的内容，0x00000这个位置原来存放着由BIOS建立的中断向量表及BIOS数据区。这个复制动作将BIOS中断向量表和BIOS数据区完全覆盖，使它们不复存在。直到新的中断服

务体系构建完毕之前，操作系统不再具备响应并处理中断的能力。现在，我们开始体会到图1-16中的关中断操作的意义。

点评

这样做能取得“一箭三雕”的效果：

- 1) 废除BIOS的中断向量表，等同于废除了BIOS提供的实模式下的中断服务程序。
- 2) 收回刚刚结束使用寿命的程序所占内存空间。
- 3) 让内核代码占据内存物理地址最开始的、天然的、有利的位置。

“破旧立新”这个成语用在这里特别贴切。

system模块复制到0x00000这个动作，废除了BIOS的中断向量表，也就是废除了16位的中断机制。操作系统是不能没有中断的，对外设的使用、系统调用、进程调度都离不开中断。Linux操作系统是32位的现代操作系统，16位的中断机制对32位的操作系统而言，显然是不合适的，这也是废除16位中断机制的根本原因。为了建立32位的操作系统，我们不但要“破旧”，还要“立新”——建立新的中断机制。

1.3.2 设置中断描述符表和全局描述符表

setup程序继续为保护模式做准备。此时要通过setup程序自身提供的数据信息对中断描述符表寄存器（IDTR）和全局描述符表寄存器（GDTR）进行初始化设置。

小贴士

GDT（Global Descriptor Table，全局描述符表），在系统中唯一的存放段寄存器内容（段描述符）的数组，配合程序进行保护模式下的段寻址。它在操作系统的进程切换中具有重要意义，可理解为所有进程的总目录表，其中存放每一个任务（task）局部描述符表（LDT,Local Descriptor Table）地址和任务状态段（TSS,Task Structure

Segment) 地址, 完成进程中各段的寻址、现场保护与现场恢复。

GDTR (Global Descriptor Table Register, GDT 基地址寄存器), GDT 可以存放在内存的任何位置。当程序通过段寄存器引用一个段描述符时, 需要取得 GDT 的入口, GDTR 标识的即为此入口。在操作系统对 GDT 的初始化完成后, 可以用 LGDT (Load GDT) 指令将 GDT 基地址加载至 GDTR。

IDT (Interrupt Descriptor Table, 中断描述符表), 保存保护模式下所有中断服务程序的入口地址, 类似于实模式下的中断向量表。

IDTR (Interrupt Descriptor Table Register, IDT 基地址寄存器), 保存IDT的起始地址。

内核实现代码如下:

```
//代码路径: boot/setup.s

.....

end _move:

mov ax, #SETUPSEG ! right, forgot this at first. didn't work: -)

mov ds, ax

lidt idt_48 ! load idt with 0, 0

lgdt gdt_48 ! load gdt with whatever appropriate

.....

gdt:

.word 0, 0, 0, 0 ! dummy

.word 0x07FF ! 8Mb-limit=2047 (2048*4096=8Mb)

.word 0x0000 ! base address=0
```

```
.word 0x9A00! code read/exec

.word 0x00C0! granularity=4096, 386

.word 0x07FF! 8Mb-limit=2047 (2048*4096=8Mb)

.word 0x0000! base address=0

.word 0x9200! data read/write

.word 0x00C0! granularity=4096, 386

idt_48:

.word 0! idt limit=0

.word 0, 0! idt base=0L

gdt_48:

.word 0x800! gdt limit=2048, 256 GDT entries

.word 512+gdt, 0x9! gdt base=0X9xxxx

.....
```

这些代码设置所需要的数据分别在idt_48和gdt_48所对应的标号处，它们和寄存器的对应方

式如图1-18所示。

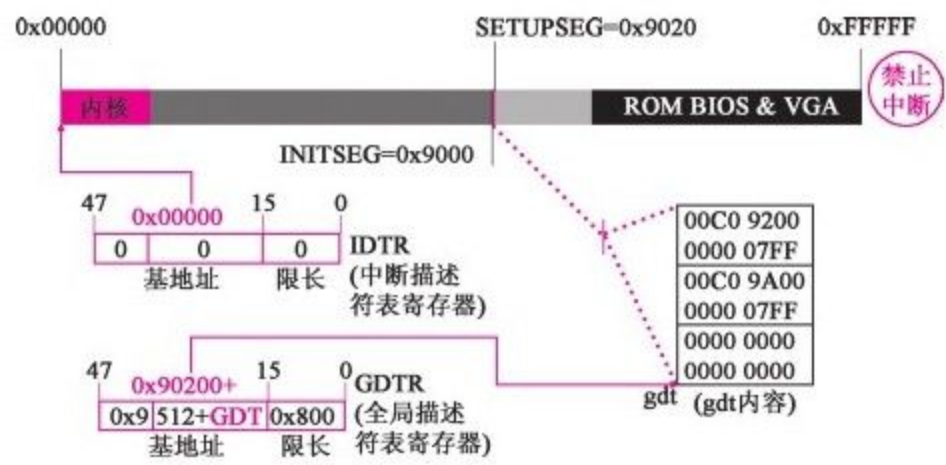


图 1-18 设置GDTR和IDTR

点评

32位的中断机制和16位的中断机制，在原理上有比较大的差别。最明显的是16位的中断机制用的是中断向量表，中断向量表的起始位置在0x00000处，这个位置是固定的；32位的中断机制用的是中断描述符表（IDT），位置是不固定

的，可以由操作系统的设计者根据设计要求灵活安排，由IDTR来锁定其位置。

GDT是保护模式下管理段描述符的数据结构，对操作系统自身的运行以及管理、调度进程有重大意义，后面的章节会有详细讲解。

因为，此时此刻内核尚未真正运行起来，还没有进程，所以现在创建的GDT第一项为空，第二项为内核代码段描述符，第三项为内核数据段描述符，其余项皆为空。

IDT虽然已经设置，实为一张空表，原因是目前已关中断，无需调用中断服务程序。此处反映的是数据“够用即得”的思想。

创建这两个表的过程可理解为是分两步进行的：

- 1) 在设计内核代码时，已经将两个表写好，并且把需要的数据也写好。

- 2) 将专用寄存器（IDTR、GDTR）指向表。

此处的数据区域是在内核源代码中设定、编译并直接加载至内存形成的一块数据区域。专用寄存器的指向由程序中的lidt和lgdt指令完成，具体操作见图1-18。

值得一提的是，在内存中做出数据的方法有两种：

1) 划分一块内存区域并初始化数据,“看住”这块内存区域,使之能被找到;

2) 由代码做出数据,如用push代码压栈,“做出”数据。

此处采用的是第一种方法。

1.3.3 打开A20，实现32位寻址

下面是标志性的动作——打开A20！

打开A20，意味着CPU可以进行32位寻址，最大寻址空间为4 GB。注意图1-19中内存条范围的变化：从5个F扩展到8个F，即0xFFFFFFFF——4 GB。



图 1-19 打开A20

现在看来，Linux 0.11还显得有些稚嫩，最大只能支持16 MB的物理内存，但是其线性寻址空间已经是不折不扣的4 GB。

打开A20的代码（boot/setup.s）如下：

//代码路径： boot/setup.s

.....

! that was painless,now we enable A20

call empty_8042

mov al, #0xD1 ! command write

out#0x64, al

call empty_8042

mov al, #0xDF ! A20 on

out#0x60, al

call empty_8042

.....

点评

实模式下CPU寻址范围为0~0xFFFFF，共1 MB寻址空间，需要0~19号共20根地址线。进入保护模式后，将使用32位寻址模式，即采用32根

地址线进行寻址，第21根（A20）至第32根地址线的选通控制将意味着寻址模式的切换。

实模式下，当程序寻址超过0xFFFFF时，CPU将“回滚”至内存地址起始处寻址（注意，在只有20根地址线的条件下， $0xFFFFF+1=0x00000$ ，最高位溢出）。例如，系统的段寄存器（如CS）的最大允许地址为0xFFFF，指令指针（IP）的最大允许段内偏移也为0xFFFF，两者确定的最大绝对地址为0x10FFEF，这将意味着程序中可产生的实模式下的寻址范围比1 MB多出将近64 KB（一些特殊寻址要求的程序就利用了这个特点）。这样，此处对A20地址线的启用相当于关闭CPU在实模式下

寻址的“回滚”机制。在后续代码中也将看到利用此特点来验证A20地址线是否确实已经打开。

1.3.4 为保护模式下执行head.s做准备

为了建立保护模式下的中断机制，`setup`程序将对可编程中断控制器8259A进行重新编程。

小贴士

8259A：专门为了对8085A和8086/8088进行中断控制而设计的芯片，是可以用程序控制的中断控制器。单个的8259A能管理8级向量优先级中断，在不增加其他电路的情况下，最多可以级联成64级的向量优先级中断系统。

具体代码如下：

```
//代码路径: boot/setup.s
```

```
.....
```

mov al, #0x11 ! initialization sequence

out#0x20, al ! send it to 8259A-1

.word 0x00eb, 0x00eb ! jmp\$+2, jmp\$+2

out#0xA0, al ! and to 8259A-2

.word 0x00eb, 0x00eb

mov al, #0x20 ! start of hardware int's (0x20)

out#0x21, al

.word 0x00eb, 0x00eb

mov al, #0x28 ! start of hardware int's 2 (0x28)

out#0xA1, al

.word 0x00eb, 0x00eb

mov al, #0x04 ! 8259-1 is master

out#0x21, al

.word 0x00eb, 0x00eb

mov al, #0x02 ! 8259-2 is slave

out#0xA1, al

.word 0x00eb, 0x00eb


```
mov al, #0x01 ! 8086 mode for both

out#0x21, al

.word 0x00eb, 0x00eb

out#0xA1, al

.word 0x00eb, 0x00eb

mov al, #0xFF ! mask off all interrupts for now

out#0x21, al

.word 0x00eb, 0x00eb

out#0xA1, al

.....
```

重新编程的结果在图1-20中有直观的表达。

CPU在保护模式下，int 0x00～int 0x1F被Intel保留作为内部（不可屏蔽）中断和异常中断。如果不对8259A进行重新编程，int 0x00～int 0x1F中断将被覆盖。例如，IRQ0（时钟中断）为8号

(int 0x08) 中断，但在保护模式下此中断号是 Intel 保留的“Double Fault”（双重故障）。因此，必须通过 8259A 编程将原来的 IRQ0x00~IRQ0x0F 对应的中断号重新分布，即在保护模式下，IRQ0x00~IRQ0x0F 的中断号是 int 0x20~int 0x2F。

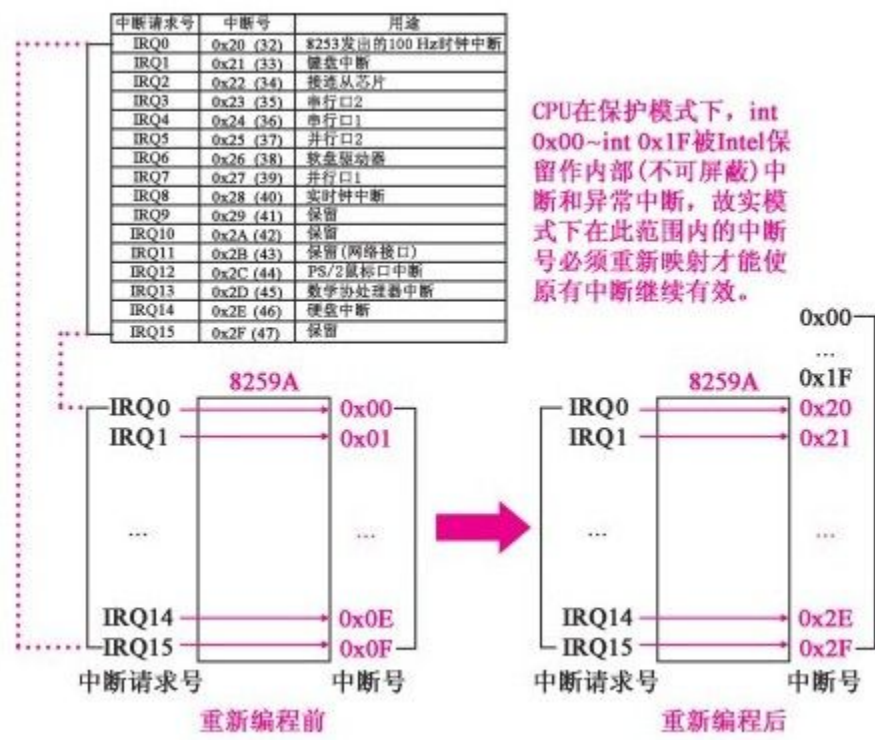


图 1-20 对可编程中断控制器重新编程

setup程序通过下面代码的前两行将CPU工作方式设为保护模式。将CR0寄存器第0位（PE）置1，即设定处理器工作方式为保护模式。

小贴士

CR0寄存器：0号32位控制寄存器，存放系统控制标志。第0位为PE（Protected Mode Enable，保护模式使能）标志，置1时CPU工作在保护模式下，置0时为实模式。

具体代码如下：

```
//代码路径: boot/setup.s
```

```
.....
```

```
mov ax, #0x0001 ! protected mode (PE) bit
```

```
lmsw ax ! This is it !
```

```
jmp 0, 8! jmp offset 0 of segment 8 (cs)
```

.....

图1-21对此做出了直观的标示。

CPU工作方式转变为保护模式，一个重要的特征就是要根据GDT决定后续执行哪里的程序。

注意看图1-18中对GDT的设置，这些设置都是setup事先安排好了的默认设置。从setup程序跳转到head程序的方式如图1-22所示。

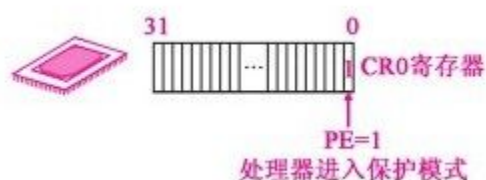


图 1-21 打开保护模式

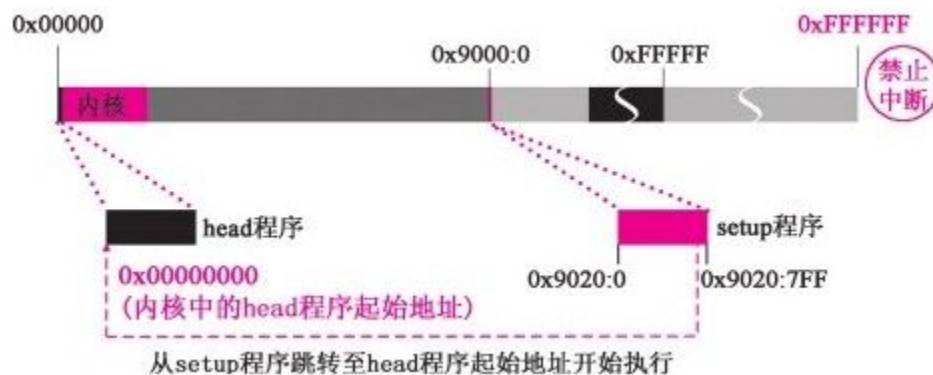


图 1-22 程序段间跳转

具体代码如下：

```
//代码路径: boot/setup.s
```

```
.....
```

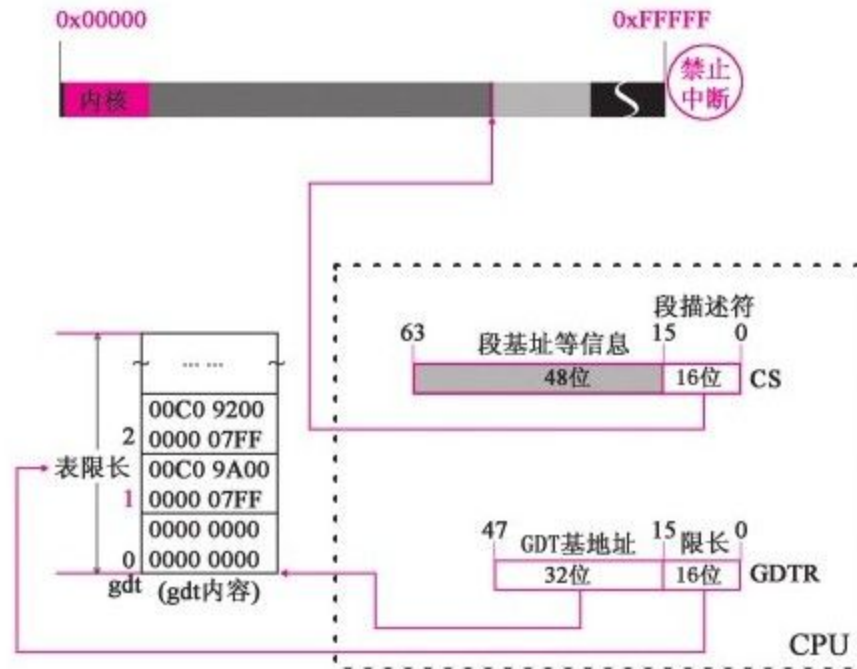
```
jmpb 0, 8
```

```
.....
```

这一行代码中的“0”是段内偏移，“8”是保护模式下的段选择符，用于选择描述符表和描述符表项以及所要求的特权级。这里“8”的解读方式很

有意思。如果把“8”当做6、7、8.....中的“8”这个数来看待，这行程序的意思就很难理解了。必须把“8”看成二进制的1000，再把前后相关的代码联合起来当做一个整体看，在头脑中形成类似图1-23所示的图，才能真正明白这行代码究竟在说什么。注意：这是一个以位为操作单位的数据使用方式，4 bit的每一位都有明确的意义，这是底层源代码的一个特点。

保护模式开启前



保护模式开启后

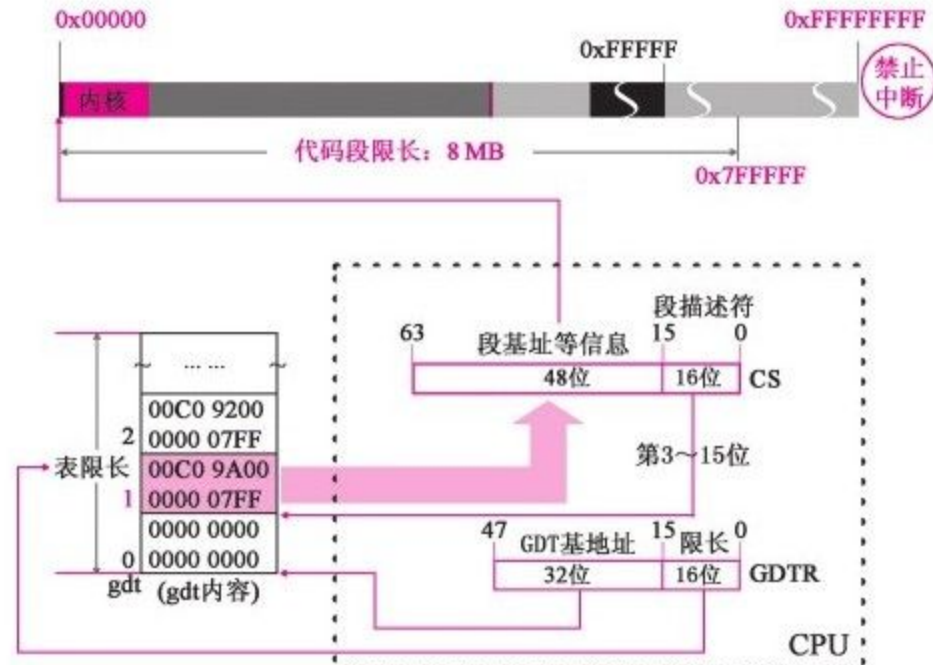


图 1-23 保护模式开启前后的指令寻址方式对比示意图

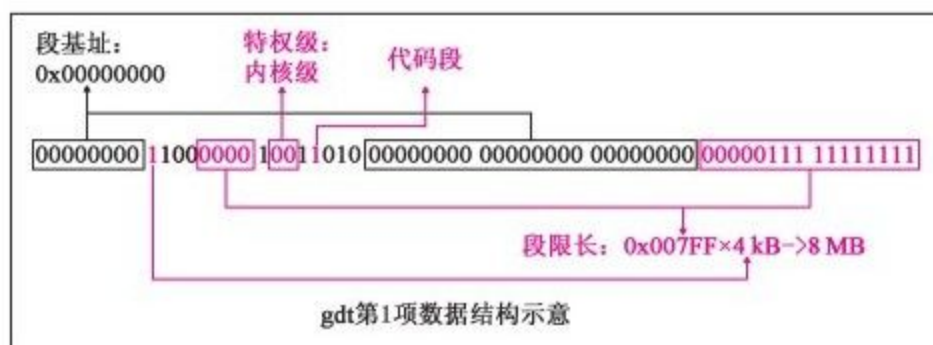


图 1-23 (续)

这里1000的最后两位（00）表示内核特权级，与之相对的用户特权级是11；第三位的0表示GDT，如果是1，则表示LDT；1000的1表示所选的表（在此就是GDT）的1项（GDT项号排序为0项、1项、2项，这里也就是第2项）来确定代码段的段基址和段限长等信息。从图1-23中我们可以看到，代码是从段基址0x00000000、偏移为0处，

也就是head程序的开始位置开始执行的，这意味着执行head程序。

1.3.5 head.s开始执行

在讲解head程序之前，我们先介绍一下从bootsect到main函数执行的整体技术策略。

在执行main函数之前，先要执行三个由汇编代码生成的程序，即bootsect、setup和head。之后，才执行由main函数开始的用C语言编写的操作系统内核程序。

前面我们讲过，第一步，加载bootsect到0x07C00，然后复制到0x90000；第二步，加载setup到0x90200。值得注意的是，这两段程序是分别加载、分别执行的。

head程序与它们的加载方式有所不同。大致的过程是，先将head.s汇编成目标代码，将用C语言编写的内核程序编译成目标代码，然后链接成system模块。也就是说，system模块里面既有内核程序，又有head程序。两者是紧挨着的。要点是，head程序在前，内核程序在后，所以head程序名字为“head”。head程序在内存中占有25 KB+184 B的空间。前面讲解过，system模块加载到内存后，setup将system模块复制到0x00000位置，由于head程序在system的前面，所以实际上，head程序就在0x00000这个位置。head程序、以main函数开始的内核程序在system模块中的布局示意图如图1-24所示。

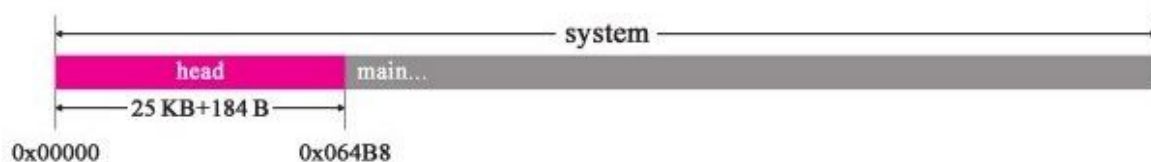


图 1-24 system在内存中的分布示意图

head程序除了做一些调用main的准备工作之外，还做了一件对内核程序在内存中的布局及内核程序的正常运行有重大意义的事，就是用程序自身的代码在程序自身所在的内存空间创建了内核分页机制，即在0x000000的位置创建了页目录表、页表、缓冲区、GDT、IDT，并将head程序已经执行过的代码所占内存空间覆盖。这意味着head程序自己将自己废弃，main函数即将开始执行。

以上就是head程序执行过程的整体策略。我们参照这个策略，看看head究竟是怎么执行的。

在讲解head程序执行之前，我们先来关注一个标号：_pg_dir，如下面的代码（boot/head.s）所

示:

```
//代码路径: boot/head.s
```

```
.....
```

```
.text
```

```
.globl _idt, _gdt, _pg_dir, _tmp_floppy_area
```

```
_pg_dir:
```

```
startup_32:
```

```
movl $0x10, %eax
```

```
mov %ax, %ds
```

```
mov %ax, %es
```

```
mov %ax, %fs
```

```
mov %ax, %gs
```

```
.....
```

标号_pg_dir标识内核分页机制完成后的内核起始位置，也就是物理内存的起始位置0x000000。head程序马上就要在此处建立页目录表，为分页机制做准备。这一点非常重要，是内核能够掌控用户进程的基础之一，后续章节将逐步讲解。图1-25中描述了页目录表在内存中所占的位置。



图 1-25 建立内核分页机制

现在head程序正式开始执行，一切都是为适应保护模式做准备。在图1-25中，其本质就是让CS的用法从实模式转变到保护模式。在实模式

下，CS本身就是代码段基址。在保护模式下，CS本身不是代码段基址，而是代码段选择符。通过对图1-25的分析得知，`jmp 0, 8`这句代码使CS和GDT的第2项关联，并且使代码段基址指向0x000000。

从现在开始，要将DS、ES、FS和GS等其他寄存器从实模式转变到保护模式。执行代码如下：

```
//代码路径: boot/head.s
```

```
.....
```

```
startup_32:
```

```
movl $0x10, %eax
```

```
mov %ax, %ds
```

```
mov %ax, %es
```

```
mov %ax, %fs
```

```
mov %ax, %gs
```

```
.....
```

执行完毕后，DS、ES、FS和GS中的值都成为0x10。与前面提到的jmp 0, 8中的8的分析方法相同，0x10也应看成二进制的00010000，最后三位与前面讲解的一样，其中最后两位（00）表示内核特权级，从后数第3位（0）表示选择GDT，第4、5两位（10）是GDT的2项，也就是第3项。也就是说，4个寄存器用的是同一个全局描述符，它们的段基址、段限长、特权级都是相同的。特别要注意的是，影响段限长的关键字段的值是0x7FF，段限长就是8 MB。

图1-26给出了详细示意。

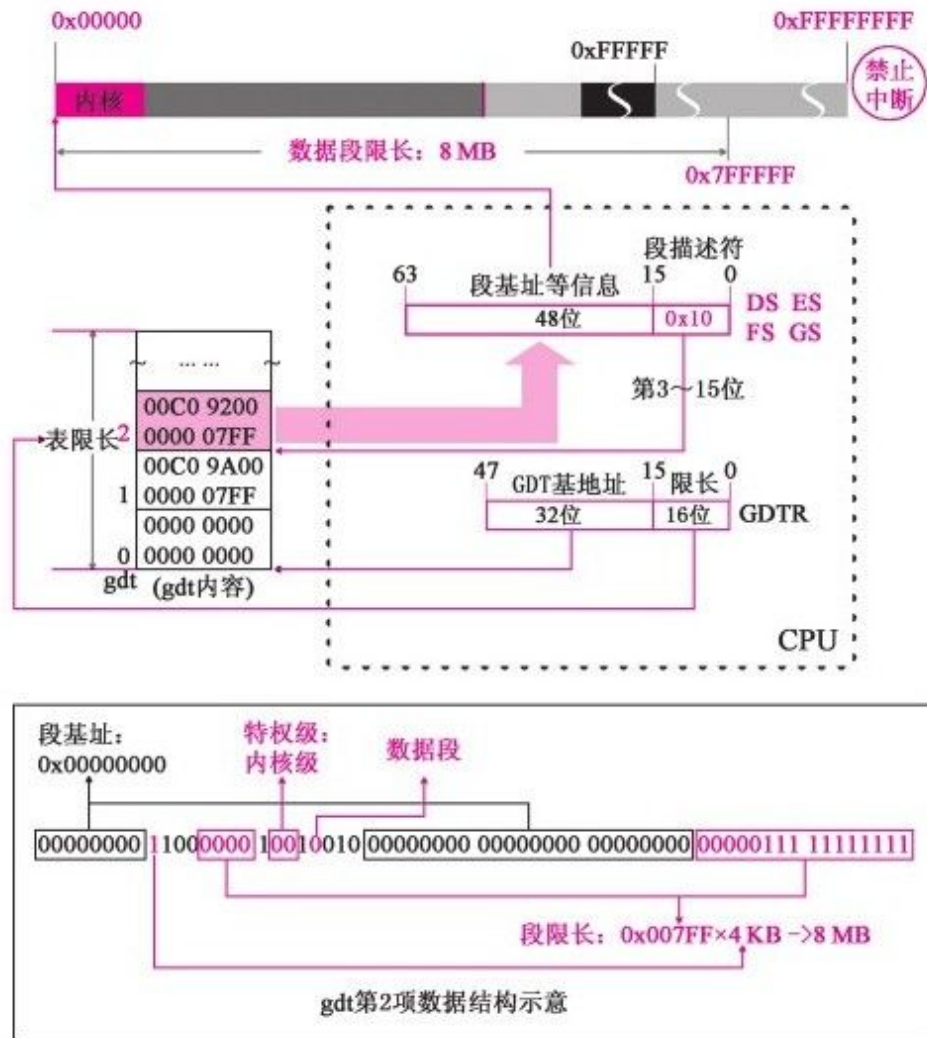


图 1-26 设置DS、ES、FS、GS

具体的设置方式与图1-23类似，都要参考GDT中的内容。上述代码中的`movl$0x10, %eax`中的0x10是GDT中的偏移值（用二进制表示就是10000），即要参考GDT中第2项的信息（GDT项

号排序为第0项、第1项、第2项) 来设置这些段寄存器，这一项就是内核数据段描述符。

点评

各段重叠，这样的编码操作方式需要头脑非常清楚！

SS现在也要转变为栈段选择符，栈顶指针也成为32位的esp，如下所示。

```
lss _stack_start, %esp
```

在kernel/sched.c中，`stack_start={ &user_stack[PAGE_SIZE >> 2], 0x10}`这行代码将栈顶指针指向user_stack数据结构的最末位置。这

个数据结构是在kernel/sched.c中定义的，如下所示：

```
long user_stack[PAGE_SIZE >> 2]
```

我们测算出其起始位置为0x1E25C。

小贴士

设置段寄存器指令（Load Segment Instruction）：该组指令的功能是把内存单元的一个“低字”传送给指令中指定的16位寄存器，把随后的一个“高字”传给相应的段寄存器（DS、ES、FS、GS和SS）。其指令格式如下：

```
LDS/LES/LFS/LGS/LSS  Mem,Reg
```

指令LDS（Load Data Segment Register）和LES（Load Extra Segment Register）在8086 CPU中就存在，而LFS和LGS、LSS（Load Stack Segment Register）是80386及其以后CPU中才有的指令。若Reg是16位寄存器，则Mem必须是32位指针；若Reg是32位寄存器，则Mem必须是48位指针，其低32位给指令中指定的寄存器，高16位给指令中的段寄存器。

0x10将SS设置为与前面4个段选择符的值相同。这样SS与前面讲解过的4个段选择符相同，段基址都是指向0x000000，段限长都是8 MB，特权级都是内核特权级，后面的压栈动作就要在这里进行。

特别值得一提的是，现在刚刚从实模式转变到保护模式，段基址的使用方法和实模式差别非常大，要使用GDT产生段基址，前面讲到的那几行设置段选择符的指令本身都是要用GDT寻址的。现在就能清楚地看出，如果没有setup程序在16位实模式下模拟32位保护模式而创建的GDT，恐怕前面这几行指令都无法执行。

注意，栈顶的增长方向是从高地址向低地址的，参见图1-27。注意栈段基址和ESP在图中的位置。

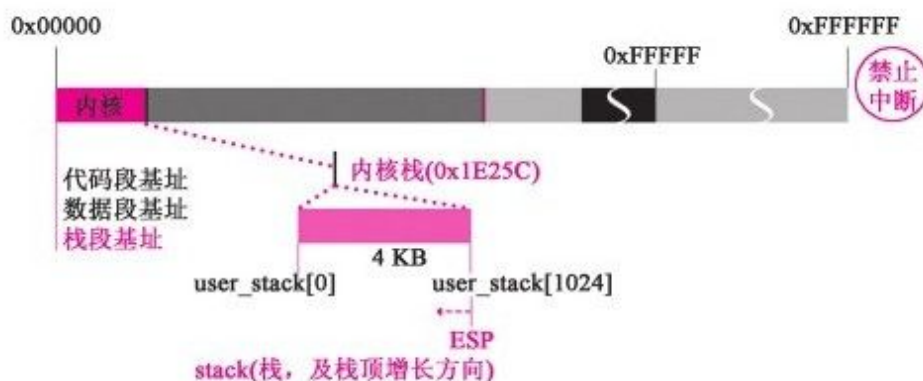


图 1-27 设置栈

我们现在回忆一下图1-8中对栈顶指针的设置，那时候是设置SP，而这时候是设置ESP，多加了一个字母E，这是为适应保护模式而做的调整。这段内容对应的代码如下：

```
//代码路径: boot/head.s
```

```
lss _stack_start, %esp
```

head程序接下来对IDT进行设置，代码如下：

```
//代码路径: boot/head.s
```

```
startup_32:
```

```
movl $0x10, %eax
```

```
mov %ax, %ds
```

```
mov %ax, %es
```

```
mov %ax, %fs
```

```
mov %ax, %gs
```

```
lss _stack_start, %esp
```

```
call setup_idt
```

```
call setup_gdt
```

```
.....
```

```
setup _idt:
```

```
lea ignore_int, %edx
```

movl \$0x00080000, %eax/*8应该看成1000，这个值在第2章初始化IDT时会用到

```
movw %dx, %ax/*selector=0x0008=cs*/
```

```
movw $0x8E00, %dx/*interrupt gate-dpl=0, present*/
```

```
lea _idt, %edi
```

```
mov $256, %ecx
```

```
rp _sidt:
```

```
movl %eax, (%edi)
```

```
movl %edx, 4(%edi)
```

```
addl $8, %edi
```

dec %ecx

jne rp_sidt

lidt idt_descr

ret

.....

.align 2

ignore _int:

pushl %eax

pushl %ecx

pushl %edx

push %ds

push %es

push %fs

movl \$0x10, %eax

mov %ax, %ds

mov %ax, %es

mov %ax, %fs


```
pushl $int_msg
```

```
call _printk
```

```
popl %eax
```

```
pop %fs
```

```
pop %es
```

```
pop %ds
```

```
popl %edx
```

```
popl %ecx
```

```
popl %eax
```

```
iret
```

```
.....
```

```
.align 2
```

```
.word 0
```

```
idt_descr:
```

```
.word 256*8-1#idt contains 256 entries
```

```
.long_idt
```

```
.....
```

```
.align 3

_idt: .fill 256, 8, 0#idt is uninitialized

.....
```

小贴士

一个中断描述符的结构如图1-28所示。



图 1-28 中断描述符

中断描述符为64位，包含了其对应中断服务程序的段内偏移地址（OFFSET）、所在段选择符

（**SELECTOR**）、描述符特权级（**DPL**）、段存在标志（**P**）、段描述符类型（**TYPE**）等信息，供CPU在程序中需要进行中断服务时找到相应的中断服务程序。其中，第0～15位和第48～63位组合成32位的中断服务程序的段内偏移地址

（**OFFSET**）；第16～31位为段选择符

（**SELECTOR**），定位中断服务程序所在段；第47位为段存在标志（**P**），用于标识此段是否存在于内存中，为虚拟存储提供支持；第45～46位为特权级标志（**DPL**），特权级范围为0～3；第40～43位为段描述符类型标志（**TYPE**），中断描述符对应的类型标志为0111（0xE），即将此段描述符标记为“386中断门”。

这是重建保护模式下中断服务体系的开始。程序先让所有的中断描述符默认指向ignore_int这

个位置（将来main函数里面还要让中断描述符对应具体的中断服务程序），之后还要对IDT寄存器的值进行设置。图1-29显示了具体的操作状态。

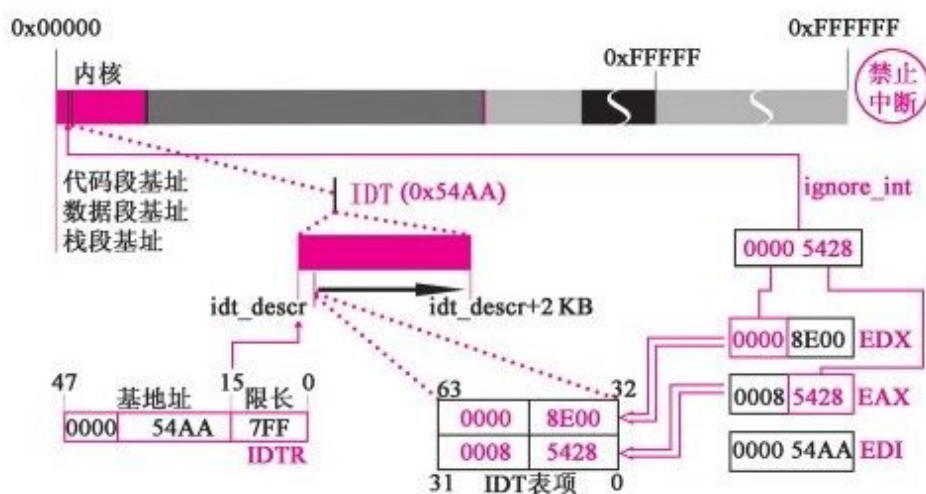


图 1-29 设置IDT

点评

构造IDT，使中断机制的整体架构先搭建起来（实际的中断服务程序挂接则在main函数中完成），并使所有中断服务程序指向同一段只显示一行提示信息就返回的服务程序。从编程技术上

讲，这种初始化操作，既可以防止无意中覆盖代码或数据而引起的逻辑混乱，也可以对开发过程中的误操作给出及时的提示。IDT有256个表项，实际只使用了几十个，对于误用未使用的中断描述符，这样的提示信息可以提醒开发人员注意错误。

现在，head程序要废除已有的GDT，并在内核中的新位置重新创建GDT，如图1-30所示。其中第2项和第3项分别为内核代码段描述符和内核数据段描述符，其段限长均被设置为16 MB，并设置GDTR的值。

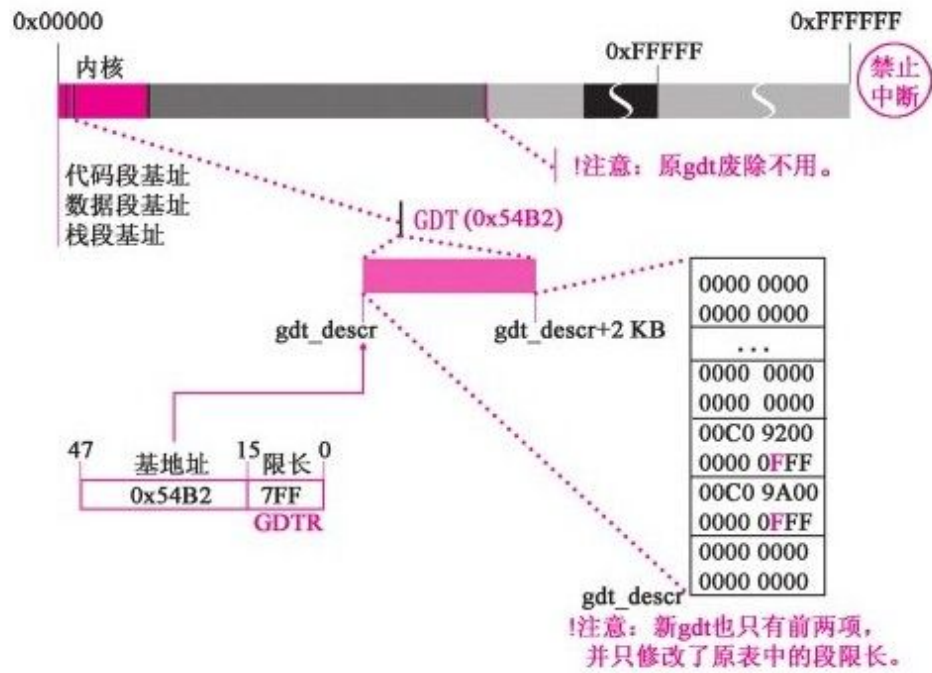


图 1-30 重新创建GDT

代码如下：

//代码路径： boot/head.s

.....

startup_32:

movl \$0x10, %eax

mov %ax, %ds

mov %ax, %es

```
mov %ax, %fs
```

```
mov %ax, %gs
```

```
lss _stack_start, %esp
```

```
call setup_idt
```

```
call setup_gdt
```

```
.....
```

```
setup _gdt:
```

```
lgdt gdt_descr
```

```
ret
```

```
.....
```

```
.align 2
```

```
.word 0
```

```
gdt _descr:
```

```
.word 256*8-1#so does gdt (not that that's any
```

```
.long _gdt#magic number,but it works for me: ^)
```

```
.....
```

```
.align 3
```

```
_idt: .fill 256, 8, 0#idt is uninitialized

_gdt: .quad 0x0000000000000000/*NULL descriptor*/

.quad 0x00c09a00000000fff/*16Mb*/

.quad 0x00c09200000000fff/*16Mb*/

.quad 0x0000000000000000/*TEMPORARY-don't use*/

.fill 252, 8, 0/*space for LDT's and TSS's etc*/
```

点评

为什么要废除原来的（GDT）而重新设置一套GDT呢？

原来GDT所在的位置是设计代码时在setup.s里面设置的数据，将来这个setup模块所在的内存位置会在设计缓冲区时被覆盖。如果不改变位置，将来GDT的内容肯定会被缓冲区覆盖掉，从而影

响系统的运行。这样一来，将来整个内存中唯一安全的地方就是现在head.s所在的位置了。

那么有没有有可能在执行setup程序时直接把GDT的内容复制到head.s所在的位置呢？肯定不能。如果先复制GDT的内容，后移动system模块，它就会被后者覆盖；如果先移动system模块，后复制GDT的内容，它又会把head.s对应的程序覆盖，而这时head.s还没有执行。所以，无论如何，都要重新建立GDT。

GDT的位置和内容发生了变化，特别要注意最后的三位是FFF，说明段限长不是原来的8 MB，而是现在的16 MB。如果后面的代码第一次使用这几个段选择符，就是访问8 MB以后的地址空间，将会产生段限长超限报警。为了防止这类

可能发生的情况，这里再次对一些段选择符进行重新设置，包括DS、ES、FS、GS及SS，方法与图1-26类似，主要是段限长增加了一倍，变为16 MB。上述过程如图1-31所示。

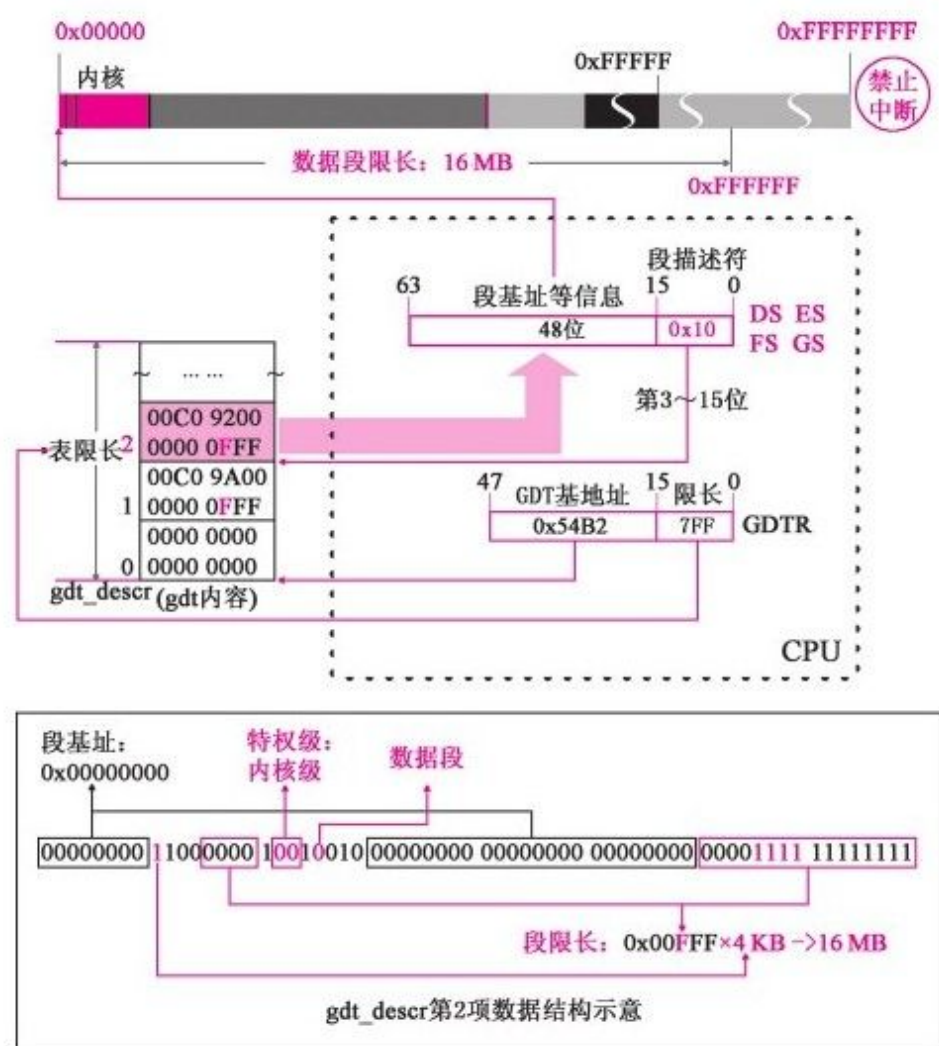


图 1-31 再一次调整DS、ES、FS、GS

调整DS、ES等寄存器的代码如下：

```
//代码路径: boot/head.s

.....

movl $0x10, %eax#reload all the segment registers

mov %ax, %ds#after changing gdt.CS was already

mov %ax, %es#reloaded in'setup_gdt'

mov %ax, %fs

mov %ax, %gs

.....
```

现在，栈顶指针esp指向user_stack数据结构的外边缘，也就是内核栈的栈底。这样，当后面的程序需要压栈时，就可以最大限度地使用栈空间。栈顶的增长方向是从高地址向低地址的，如图1-32所示。设置esp的代码如下：

```
//代码路径: boot/head.s
```

```
.....
```

```
lss _stack_start, %esp
```

```
.....
```

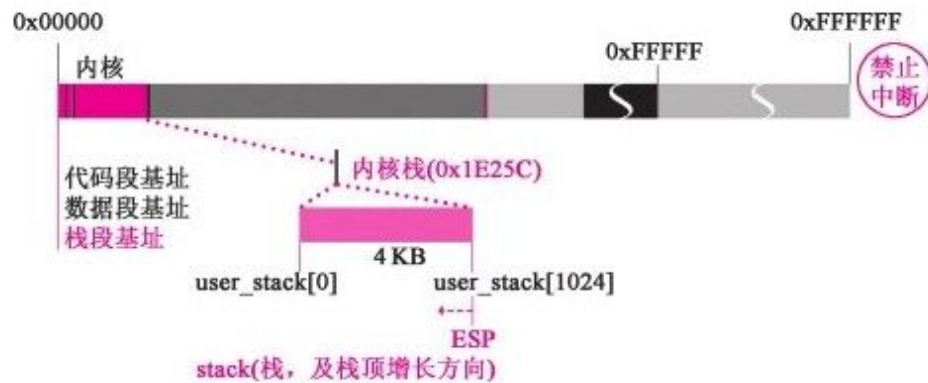


图 1-32 设置内核栈

因为A20地址线是否打开影响保护模式是否有效，所以，要检验A20地址线是否确实打开了。图1-33给出了直观的标示。



图 1-33 检验A20是否打开

检验A20是否打开的代码如下：

```
//代码路径: boot/head.s
```

```
.....
```

```
xorl %eax, %eax
```

```
1: incl%eax#check that A20 really IS enabled
```

```
movl %eax, 0x000000#loop forever if it isn't
```

```
cmpl %eax, 0x100000
```

```
je 1b
```

```
.....
```

点评

A20如果没打开，则计算机处于20位的寻址模式，超过0xFFFFF寻址必然“回滚”。一个特例是0x100000会回滚到0x000000，也就是说，地址0x100000处存储的值必然和地址0x000000处存储的值完全相同（参见对图1-31的描述）。通过在内存0x000000位置写入一个数据，然后比较此处和1 MB（0x100000，注意，已超过实模式寻址范围）处数据是否一致，就可以检验A20地址线是否已打开。

确定A20地址线已经打开之后，head程序如果检测到数学协处理器存在，则将其设置为保护模式工作状态，如图1-34所示。

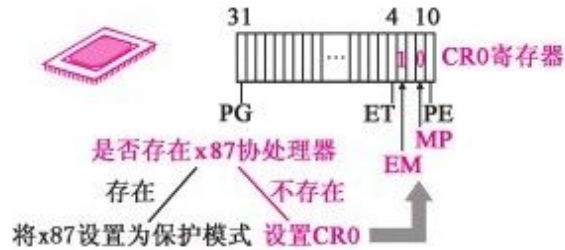


图 1-34 检测数学协处理器

小贴士

x87协处理器：为了弥补x86系列在进行浮点运算时的不足，Intel于1980年推出了x87系列数学协处理器，那时是一个外置的、可选的芯片（笔者当时的80386计算机上就没安装80387协处理器）。1989年，Intel发布了486处理器。自从486开始，以后的CPU一般都内置了协处理器。这样，对于486以前的计算机而言，操作系统检验x87协处理器是否存在就非常必要了。

检测数学协处理器对应的代码如下：

//代码路径: boot/head.s

.....

movl %cr0, %eax#check math chip

andl \$0x80000011, %eax#Save PG,PE,ET

/*"orl \$0x10020, %eax"here for 486 might be good*/

orl \$2, %eax#set MP

movl %eax, %cr0

call check_x87

jmp after_page_tables

/*

We depend on ET to be correct.This checks for 287/387./*

check_x87:

fninit

fstsw %ax

cmpb \$0, %al

je 1f/*no coprocessor: have to set bits*/


```
movl %cr0, %eax
```

```
xorl $6, %eax/*reset MP,set EM*/
```

```
movl %eax, %cr0
```

```
ret
```

```
.align 2
```

```
1: .byte 0xDB, 0xE4/*fsetpm for 287, ignored by 387*/
```

```
Ret
```

```
.....
```

head程序将为调用main函数做最后的准备。
这是head程序执行的最后阶段，也是main函数执行前的最后阶段。具体如图1-35所示。

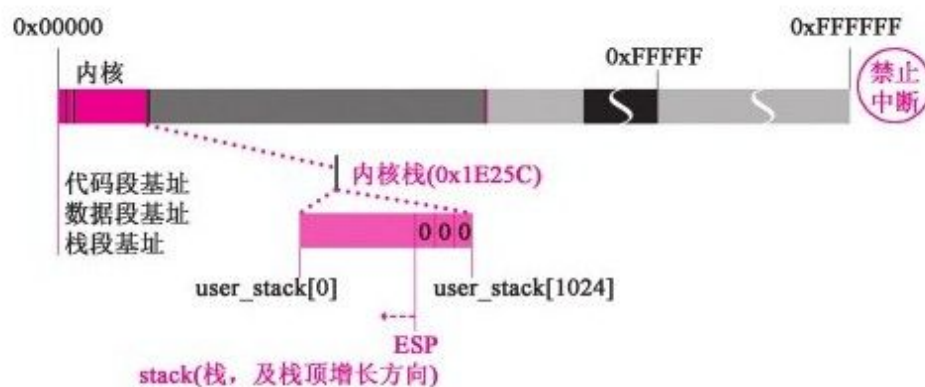


图 1-35 将envp、argv、argc压栈

head程序将L6标号和main函数入口地址压栈，栈顶为main函数地址，目的是使head程序执行完后通过ret指令就可以直接执行main函数。具体如图1-36所示。

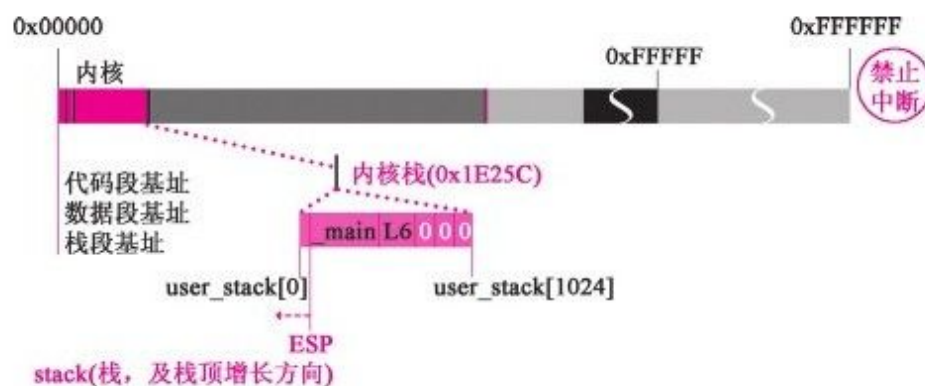


图 1-36 将main函数入口地址和L6标号压栈

点评

main函数在正常情况下是不应该退出的。如果main函数异常退出，就会返回这里的标号L6处

继续执行，此时，还可以做一些系统调用.....另外有意思的是，即使main函数退出了，如果还有进程存在，仍然能够进行轮转。

执行代码如下：

```
//代码路径: boot/head.s

.....

orl $2, %eax#set MP

movl %eax, %cr0

call check_x87

jmp after_page_tables

.....

after_page_tables:

pushl $0#These are the parameters to main: -)

pushl $0

pushl $0
```

```
pushl $L6#return address for main,if it decides to.
```

```
pushl $_main
```

```
jmp setup_paging
```

```
L6:
```

```
jmp L6#main should never return here,but
```

```
.....
```

这些压栈动作完成后，`head`程序将跳转至 `setup_paging`：去执行，开始创建分页机制。

先要将页目录表和4个页表放在物理内存的起始位置，从内存起始位置开始的5页空间内容全部清零（每页4 KB），为初始化页目录和页表做准备。注意，这个动作起到了用1个页目录表和4个页表覆盖`head`程序自身所占内存空间的作用。图1-37给出了直观的标示。

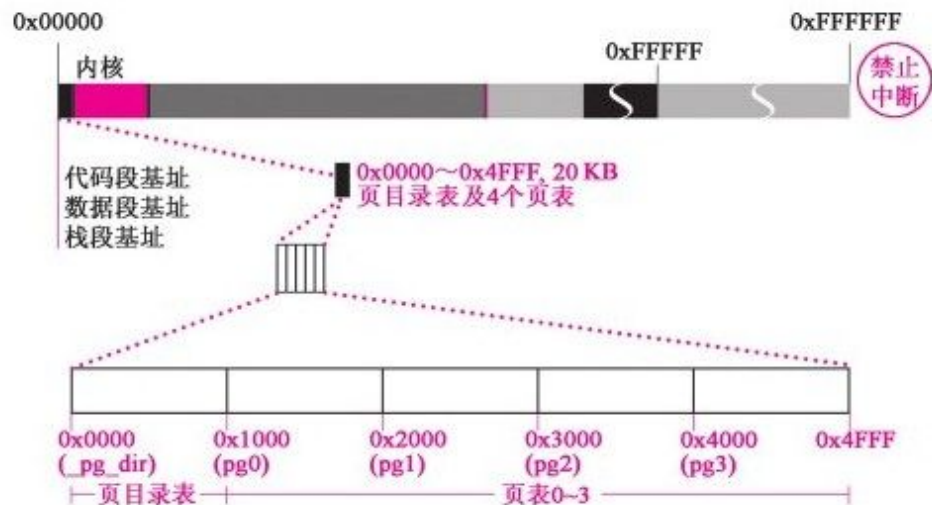


图 1-37 将页目录表和页表放在内存起始位置

点评

将页目录表和4个页表放在物理内存的起始位置，这个动作的意义重大，是操作系统能够掌控全局、掌控进程在内存中安全运行的基石之一，后续章节会逐步论述。

head程序将页目录表和4个页表所占物理内存空间清零后，设置页目录表的前4项，使之分别指

向4个页表，如图1-38所示。

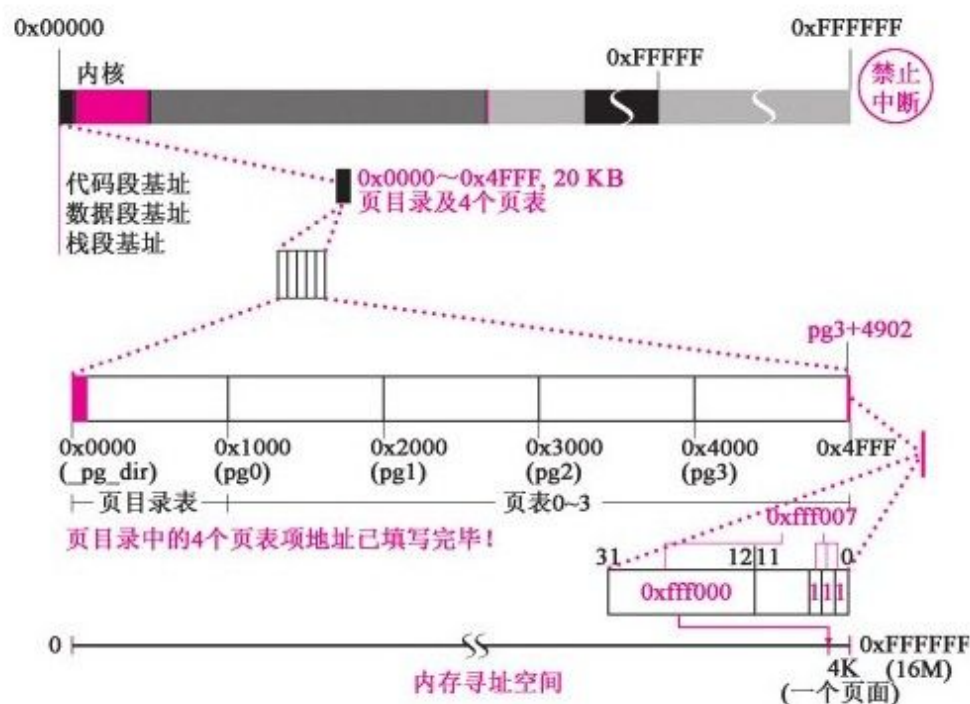


图 1-38 使页目录表的前4项指向4个页表

head程序设置完页目录表后，Linux 0.11在保护模式下支持的最大寻址地址为0xFFFFFFFF（16 MB），此处将第4个页表（由pg3指向的位置）的最后一个页表项（pg3+4902指向的位置）指向寻

址范围的最后一个页面，即0xFFF000开始的4 KB字节大小的内存空间。具体请看图1-39的标示。

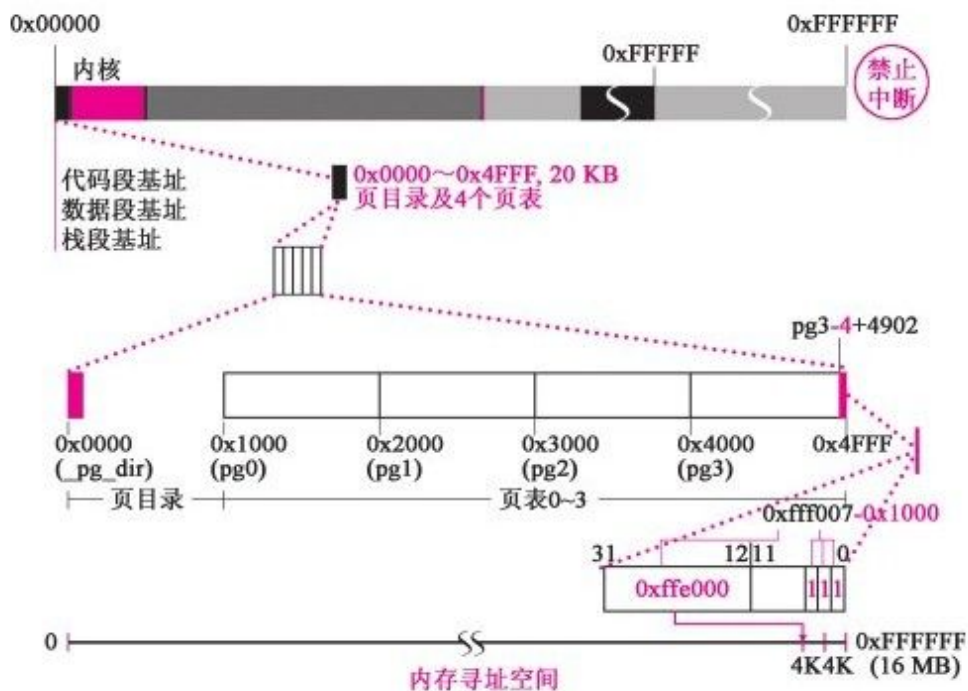


图 1-39 页目录表设置完成后的状态

然后开始从高地址向低地址方向填写4个页表，依次指向内存从高地址向低地址方向的各个页面。图1-39所示是首次设置页表。

继续设置页表。将第4个页表（由pg3指向的位置）的倒数第二个页表项（pg3-4+4902指向的位置）指向倒数第二个页面，即0xFFF000～0x1000（0x1000即4 KB，一个页面的大小）开始的4 KB字节内存空间。请读者认真对比图1-40和图1-39，有多处位置发生了变化。

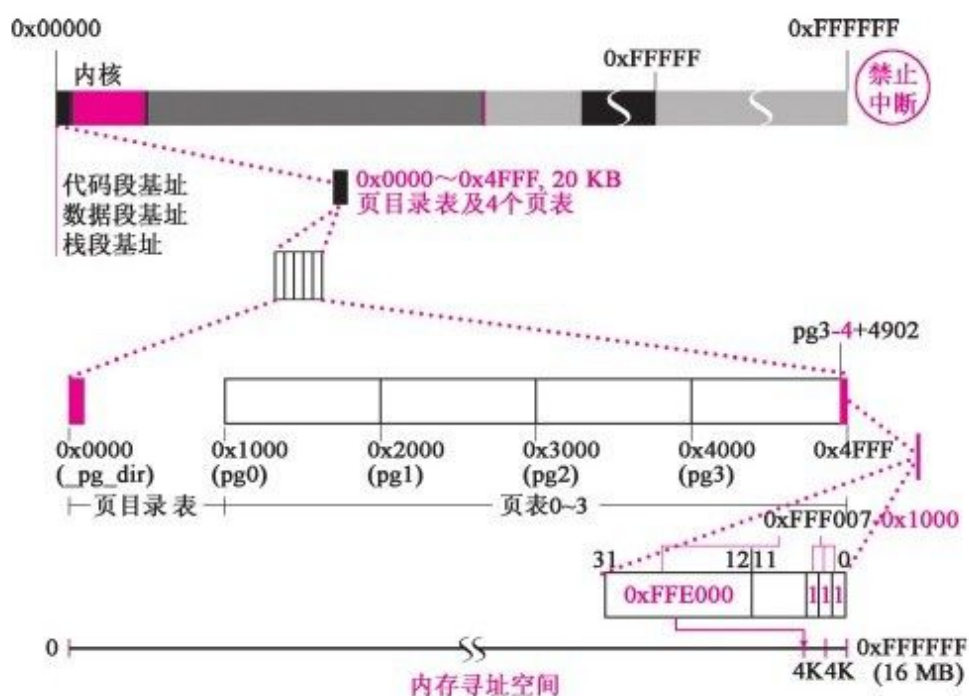


图 1-40 设置页表

最终，从高地址向低地址方向完成4个页表的填写，页表中的每一个页表项分别指向内存从高地址向低地址方向的各个页面，如图1-41所示。其总体效果如图1-42所示。

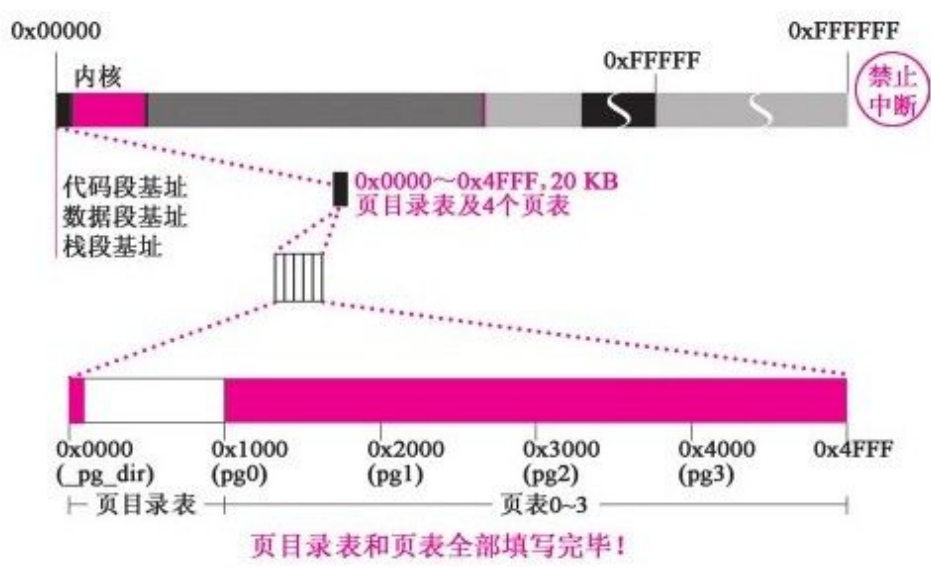


图 1-41 页目录表和页表设置完毕的状态

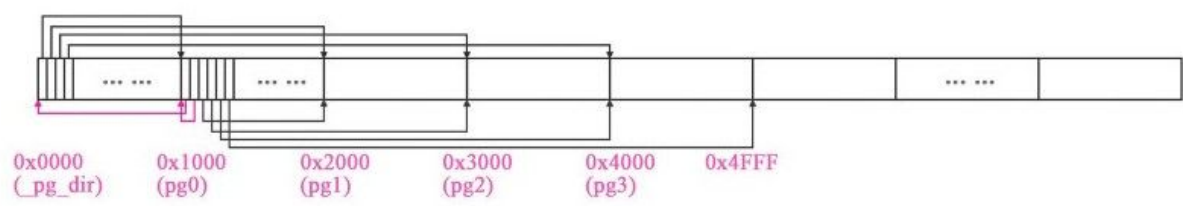


图 1-42 总体效果图

这4个页表都是内核专属的页表，将来每个用户进程都会有它们专属的页表。对于两者在寻址范围方面的区别，我们将在用户进程与内存管理一章中详细介绍。

图1-39～图1-41中发生动作的相应代码如下：

```
//代码路径: boot/head.s
```

```
.....
```

```
.align 2
```

```
setup _paging:
```

```
movl $1024*5, %ecx/*5 pages-pg_dir+4 page tables*/
```

```
xorl %eax, %eax
```

```
xorl %edi, %edi/*pg_dir is at 0x000*/
```

```
cld; rep; stosl
```

```
/*下面几行中的7应看成二进制的111，是页属性，代表u/s、r/w、present,
```

111代表：用户u、读写rw、存在p，000代表：内核s、只读r、不存在*/

```
movl $pg0+7, _pg_dir/*set present bit/user r/w*/
```

```
movl $pg1+7, _pg_dir+4/*-----"-----*/
```

```
movl $pg2+7, _pg_dir+8/*-----"-----*/
```

```
movl $pg3+7, _pg_dir+12/*-----"-----*/
```

```
movl $pg3+4092, %edi
```

```
movl $0xfff007, %eax/*16Mb-4096+7 (r/w user,p) */
```

```
std
```

```
1: stosl/*fill pages backwards-more efficient: -) */
```

```
subl $0x1000, %eax
```

```
jge 1b
```

```
.....
```

这些工作完成后，内存中的布局如图1-43所示。可以看出，只有184字节的剩余代码。由此可

见，在设计head程序和system模块时，其计算是非常精确的，对head.s的代码量的控制非常到位。

head程序已将页表设置完毕了，但分页机制的建立还没有完成，还需要设置页目录表基址寄存器CR3，使之指向页目录表，再将CR0寄存器设置的最高位（31位）置为1，如图1-44所示。



图 1-43 内存分布示意图

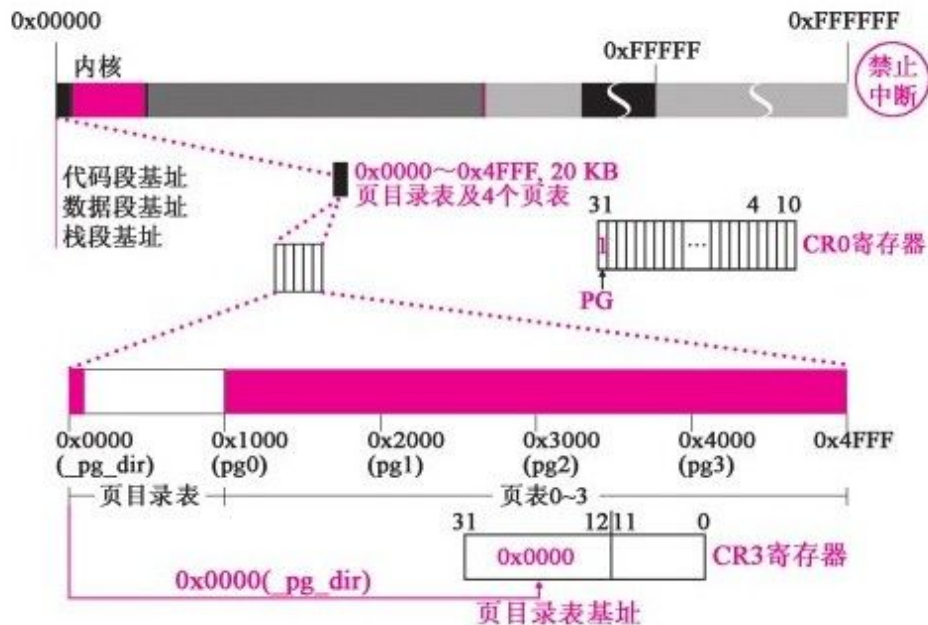


图 1-44 分页机制完成后的总体状态

小贴士

PG (Paging) 标志: CR0寄存器的第31位, 分页机制控制位。当CPU的控制寄存器CR0第0位PE (保护模式) 置为1时, 可设置PG位为开启。当开启后, 地址映射模式采取分页机制。当CPU的控制寄存器CR0第0位PE (保护模式) 置为0时, 设置PG位将引起CPU发生异常。

CR3寄存器：3号32位控制寄存器，其高20位存放页目录表的基地址。当CR0中的PG标志置位时，CPU使用CR3指向的页目录表和页表进行虚拟地址到物理地址的映射。

执行代码如下：

```
//代码路径: boot/head.s

.....

xorl %eax, %eax/*pg_dir is at 0x0000*/

movl %eax, %cr3/*cr3-page directory start*/

movl %cr0, %eax

orl $0x80000000, %eax

movl %eax, %cr0/*set paging (PG) bit*/

.....
```

前两行代码的动作是将CR3指向页目录表，意味着操作系统认定0x0000这个位置就是页目录表的起始位置；后3行代码的动作是启动分页机制开关PG标志置位，以启用分页寻址模式。两个动作一气呵成。到这里为止，内核的分页机制构建完毕。后续章节还会讲解如何建立用户进程的分页机制。

最重要的是下面这一行代码。它看似简单，但用意深远。

```
xorl %eax, %eax/*pg_dir is at 0x0000*/
```

回过头来看，图1-17将system模块移动到0x000000处，图1-25在内存的起始位置建立内核分页机制，最后就是上面的这行代码，认定页目录

表在内存的起始位置。三个动作联合起来为操作系统中最重要的目的——内核控制用户程序奠定了基础。这个位置是内核通过分页机制能够实现线性地址等于物理地址的唯一起始位置。我们会在后续章节逐层展开讨论。

head程序执行最后一步：`ret`。这要通过跳入`main`函数程序执行。

在图1-36中，`main`函数的入口地址被压入了栈顶。现在执行`ret`了，正好将压入的`main`函数的执行入口地址弹出给EIP。图1-45标示了出栈动作。

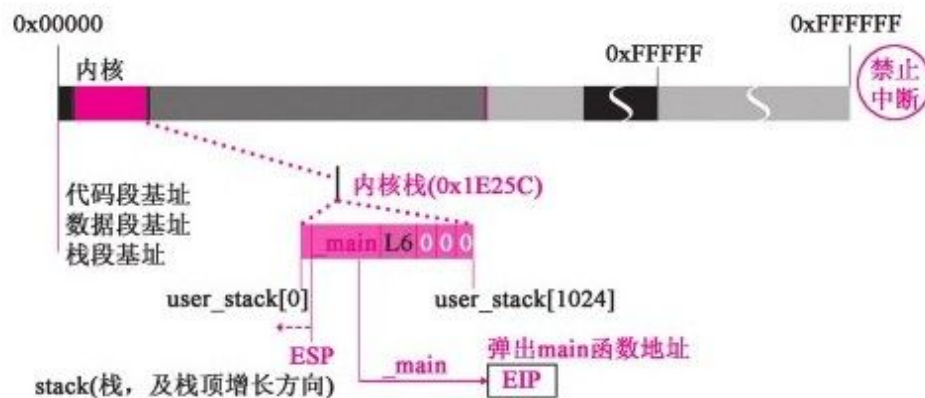
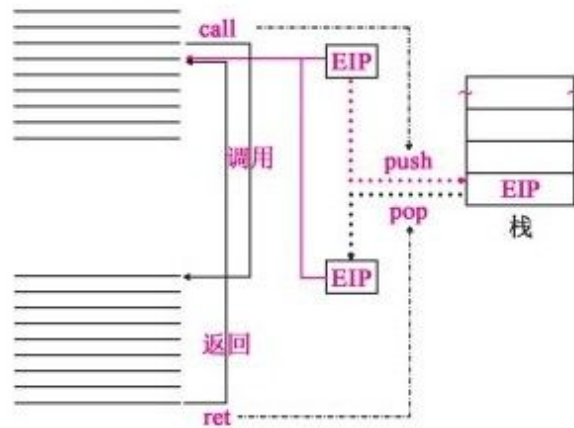


图 1-45 执行ret，将main函数入口地址弹出给EIP

这部分代码用了底层代码才会使用的技巧。我们结合图1-45对这个技巧进行详细讲解。我们先看看普通函数的调用和返回的方法。因为Linux 0.11用返回方法调用main函数，返回位置和main函数的入口在同一段内，所示我们只讲解段内调用和返回，如图1-46（仿call示意图）所示。

call的调用与返回



“仿call”的“调用”与“返回”

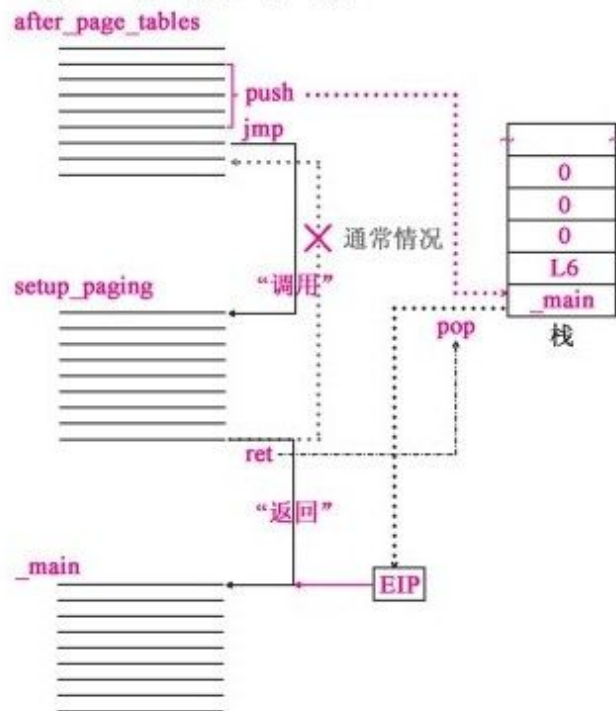


图 1-46 仿call示意图

call指令会将EIP的值自动压栈，保护返回现场，然后执行被调函数的程序。等到执行被调函数的ret指令时，自动出栈给EIP并还原现场，继续执行call的下一行指令。这是通常的函数调用方法。对操作系统的main函数来说，这个方法就有些怪异了。main函数是操作系统的。如果用call调用操作系统的main函数，那么ret时返回给谁呢？难道还有一个更底层的系统程序接收操作系统的返回吗？操作系统已经是最底层的系统了，所以逻辑上不成立。那么如何既调用了操作系统的main函数，又不需要返回呢？操作系统的设计者采用了图1-46（仿call示意图）所示的方法。

这个方法的妙处在于，是用ret实现的调用操作系统的main函数。既然是ret调用，当然就不需要再用ret了。不过，call做的压栈和跳转的动作谁

来做呢？操作系统的设计者做了一个仿call的动作，手工编写代码压栈和跳转，模仿了call的全部动作，实现了调用setup_paging函数。注意，压栈的EIP值并不是调用setup_paging函数的下一行指令的地址，而是操作系统的main函数的执行入口地址_main。这样，当setup_paging函数执行到ret时，从栈中将操作系统的main函数的执行入口地址_main自动出栈给EIP,EIP指向main函数的入口地址，实现了用返回指令“调用”main函数。

在图1-46中，将压入的main函数的执行入口地址弹出给CS: EIP，这句话等价于CPU开始执行main函数程序。图1-47标示了这个状态。

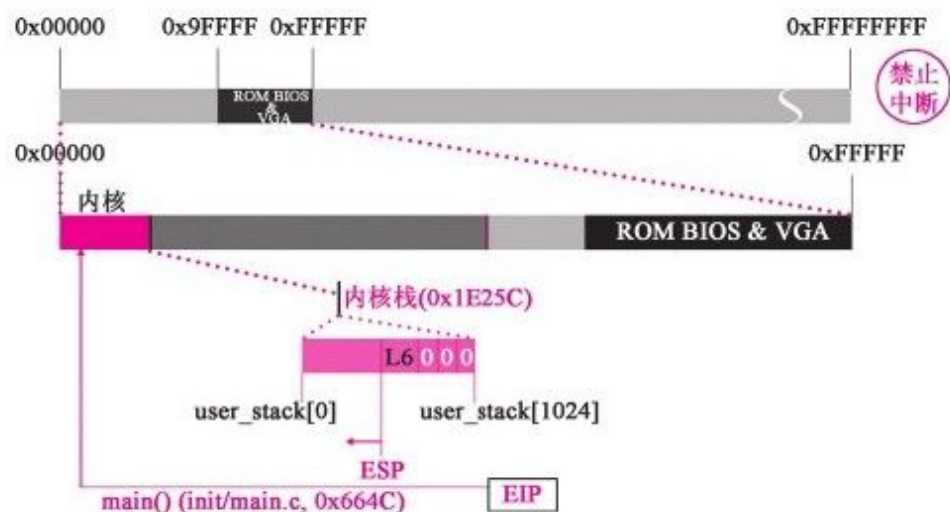


图 1-47 开始执行main函数

点评

为什么没有最先调用main函数？

学过C语言的人都知道，用C语言设计的程序都有一个main函数，而且是从main函数开始执行的。Linux 0.11的代码是用C语言编写的。奇怪的是，为什么在操作系统启动时先执行的是三个由汇编语言写成的程序，然后才开始执行main函

数；为什么不是像我们熟知的C语言程序那样，从main函数开始执行呢。

通常，我们用C语言编写的程序都是用户应用程序。这类程序的执行有一个重要的特征，就是必须在操作系统的平台上执行，也就是说，要由操作系统为应用程序创建进程，并把应用程序的可执行代码从硬盘加载到内存。现在我们讨论的是操作系统，不是普通的应用程序，这样就出现了一个问题：应用程序是由操作系统加载的，操作系统该由谁加载呢？

从前面的节中我们知道，加载操作系统的时候，计算机刚刚加电，只有BIOS程序在运行，而且此时计算机处在16位实模式状态，通过BIOS程序自身的代码形成的16位的中断向量表及相关的

16位的中断服务程序，将操作系统在软盘上的第一扇区（512字节）的代码加载到内存，BIOS能主动操作的内容也就到此为止了。准确地说，这是一个约定。对于第一扇区代码的加载，不论是什么操作系统都是一样的；从第二扇区开始，就要由第一扇区中的代码来完成后续的代码加载工作。

当加载工作完成后，好像仍然没有立即执行main函数，而是打开A20，打开pe、pg，建立IDT、GDT.....然后才开始执行main函数，这是什么道理？

原因是，Linux 0.11是一个32位的实时多任务的现代操作系统，main函数肯定要执行的是32位的代码。编译操作系统代码时，是有16位和32位

不同的编译选项的。如果选了16位，C语言编译出来的代码是16位模式的，结果可能是一个int型变量，只有2字节，而不是32位的4字节.....这不是Linux 0.11想要的。Linux 0.11要的是32位的编译结果。只有这样才能成为32位的操作系统代码。这样的代码才能用到32位总线（打开A20后的总线），才能用到保护模式和分页，才能成为32位的实时多任务的现代操作系统。

开机时的16位实模式与main函数执行需要的32位保护模式之间有很大的差距，这个差距谁来填补？head.s做的就是这项工作。这期间，head程序打开A20，打开pe、pg，废弃旧的、16位的中断响应机制，建立新的32位的IDT.....这些工作都做完了，计算机已经处在32位的保护模式状态了，调用32位main函数的一切条件已经准备完毕，这

时顺理成章地调用main函数。后面的操作就可以用32位编译的main函数完成。

至此，Linux 0.11内核启动的一个重要阶段已经完成，接下来就要进入main函数对应的代码了。

特别需要提示的是，此时仍处在关闭中断的状态！

1.4 本章小结

本章的内容主要分为两大部分。第一部分为加载操作系统；第二部分为32位保护、分页模式下的main函数的执行做准备。

从借助BIOS将bootsect.s文件加载到内存开始，相继加载了setup.s文件和system文件，从而完成操作系统程序的加载。

接下来设置IDT、GDT、页目录表、页表以及机器系统数据，为32位保护、分页模式下的main函数的执行做准备。

一切就绪后，跳转到main函数执行入口，开始执行main函数。

第2章 设备环境初始化及激活进程0

从现在开始执行main（）函数！

系统达到怠速 [1] 状态前所做的一切准备工作的核心目的就是让用户程序能够以“进程”的方式正常运行。能够实现这一目的的标准包括三方面的内容：用户程序能够在主机上进行运算，能够与外设进行交互，以及能够让用户以它为媒介进行人机交互。本章讲解的内容就是为了实现这个目标，对设备环境进行初始化，并激活第一个进程——进程0。

Linux 0.11是一个支持多进程的现代操作系统。这就意味着，各个用户进程在运行过程中，彼此不能相互干扰，这样才能保证进程在主机中

正常地运算。然而，进程自身并没有一个天然的“边界”来对其进行保护，要靠系统“人为”地给它设计一套“边界”来对其进行保护。这套“边界”就是系统为进程提供的进程管理信息数据结构。进程管理信息数据结构包括：task_struct、task[64]、GDT等。task_struct是每个进程所独有的结构。它标识了进程的各项属性值，包括剩余时间片、进程执行状态、局部数据描述符表（LDT）和任务状态描述符表（TSS）等。

task[64]和GDT是为管理多进程提供的数据结构。task[64]结构中存储着系统中所有进程的task_struct指针。如果操作系统需要对多个进程加以比较并选择，就可以通过遍历task[64]结构来实现。GDT中存储着一套针对所有进程的索引结

构。通过索引项，操作系统可以间接地与每个进程中的LDT和TSS建立关系。

本章还将讲解操作系统是如何对内存、CPU、串行口、显示器、键盘、硬盘、软盘等硬件进行设置，并将这些硬件所对应的中断服务程序与IDT相挂接，为进程0及其直接、间接创建的所有后续进程与外设沟通构建环境。

2.1 设置根设备 [2]、硬盘

内核首先初始化根设备和硬盘，用bootsect中写入机器系统数据0x901FC（见1.2.3节）的根设备为软盘的信息，设置软盘为根设备，并用起始自0x90080的32字节的机器系统数据的硬盘参数表设置内核中的硬盘信息drive_info。

具体执行代码如下：

```
//代码路径： init/main.c:
```

```
.....
```

```
#define DRIVE_INFO (* (struct drive_info *) 0x90080) //硬盘参  
数表，参看机器系统数据
```

```
#define ORIG_ROOT_DEV (* (unsigned short *) 0x901FC) //根  
设备号
```

```
.....
```

```
struct drive_info{char dummy[32]; }drive_info; //存放硬盘参数表  
的数据结构
```

```
void main (void)
```

```
{
```

```
    ROOT_DEV=ORIG_ROOT_DEV; //根据bootsect中写入机器系统  
数据的信息设置根设备为软盘
```

```
    drive_info=DRIVE_INFO; //的信息，设置为根设备
```

```
.....
```

```
}
```

设置根设备为软盘以及设置硬盘参数表完成后的数据在内存中的位置如图2-1所示。

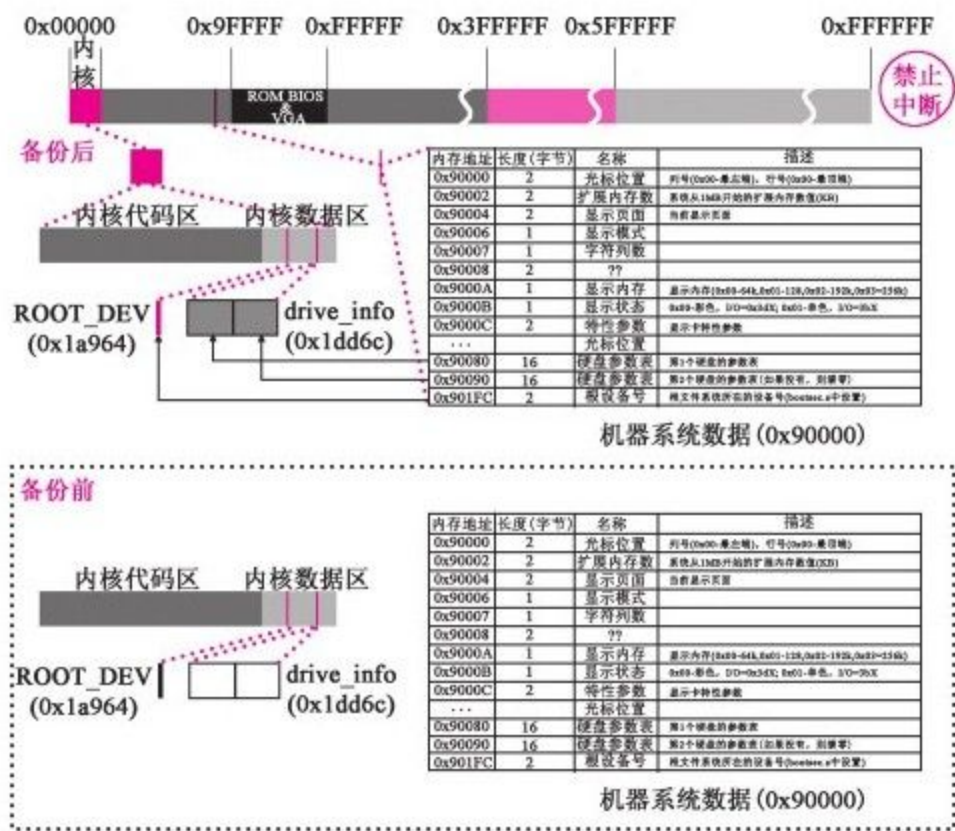


图 2-1 设置根设备号和硬盘参数表

[1] 怠速的意思就是操作系统已经完成了所有的准备工作，随时可以响应用户的激励。此处借用了汽车的“怠速”一词，是为了更加形象。汽车进入

怠速状态，就意味着汽车已经完全启动，只要驾驶员踩油门，就可以正常行驶了。

[2] 根设备就是根文件系统所在的设备，详细内容参看第3章的加载根文件系统一节。

2.2 规划物理内存格局，设置缓冲区、虚拟盘、主内存

接下来设置缓冲区、虚拟盘、主内存。主机中的运算需要CPU、内存相互配合工作才能实现，内存也是参与运算的重要部件。对内存中缓冲区、主内存的设置、规划，从根本上决定了所有进程使用内存的数量和方式，必然会影响到进程在主机中的运算速度。

具体规划如下：除内核代码和数据所占的内存空间之外，其余物理内存主要分为三部分，分别是主内存区、缓冲区和虚拟盘。主内存区是进程代码运行的空间，也包括内核管理进程的数据结构；缓冲区主要作为主机与外设进行数据交互

的中转站；“虚拟盘区”是一个可选的区域，如果选择使用虚拟盘，就可以将外设上的数据先复制进虚拟盘区，然后加以使用。由于从内存中操作数据的速度远高于外设，因此这样可以提高系统执行效率。

这里，系统要对主内存中的这三种不同性质的区域，在大小、位置以及管理方式方面进行规划。

先根据内存大小对缓冲区和主内存区的位置和大小进行如图2-2所示的初步设置。

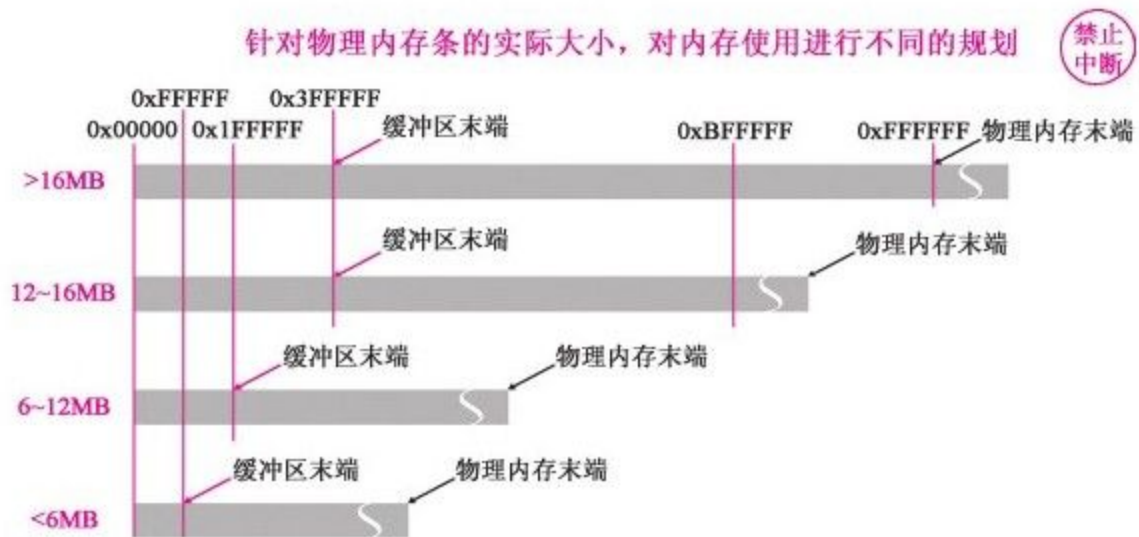


图 2-2 内存的初步设置

具体执行代码如下：

//代码路径：init/main.c:

.....

```
#define EXT_MEM_K (* (unsigned short *) 0x90002) //从1 MB  
开始的扩展内存 (KB) 数
```

.....

```
void main (void)
```

```
{
```

.....

```
memory_end= (1<<20) + (EXT_MEM_K<<10) ; //1
MB+扩展内存 (MB) 数, 即内存总数
```

```
memory_end&=0xffff000; //按页的倍数取整, 忽略内存末端不
足一页的部分
```

```
if (memory_end>16*1024*1024)
```

```
memory_end=16*1024*1024;
```

```
if (memory_end>12*1024*1024)
```

```
buffer_memory_end=4*1024*1024;
```

```
else if (memory_end>6*1024*1024)
```

```
buffer_memory_end=2*1024*1024;
```

```
else
```

```
buffer_memory_end=1*1024*1024;
```

```
main_memory_start=buffer_memory_end; //缓冲区之后就是主内
存
```

```
.....
```

```
}
```

其中memory_end为系统有效内存末端位置。超过这个位置的内存部分，在操作系统中不可见。main_memory_start为主内存区起始位置。buffer_memory_end为缓冲区末端位置。对于缓冲区的起始位置，我们将在2.10节中详细介绍。

小贴士

有几个常用的左移、右移的数据关系需要记住：

$\ll 20$ 或 $\gg 20$ 相当于乘或除以1 MB，

$\ll 12$ 或 $\gg 12$ 相当于乘或除以4 KB（联想到页），

$\ll 10$ 或 $\gg 10$ 相当于乘或除以1 KB。

所以， $1 \ll 20$ 就是1 MB, $\text{EXT_MEM_K} \ll 10$ 就是EXT_MEM_K（扩展内存的KB数）的字节数。

2.3 设置虚拟盘空间并初始化

接下来将对外设中的虚拟盘区进行设置。检查makefile文件中“虚拟盘使用标志”是否设置，以此确定本系统是否使用了虚拟盘。我们设定本书所用计算机有16 MB的内存，有虚拟盘，且将虚拟盘大小设置为2 MB。操作系统从缓冲区的末端起开辟2 MB内存空间设置为虚拟盘，主内存起始位置后移2 MB至虚拟盘的末端。图2-3展示了设置完成后的物理内存的规划格局。

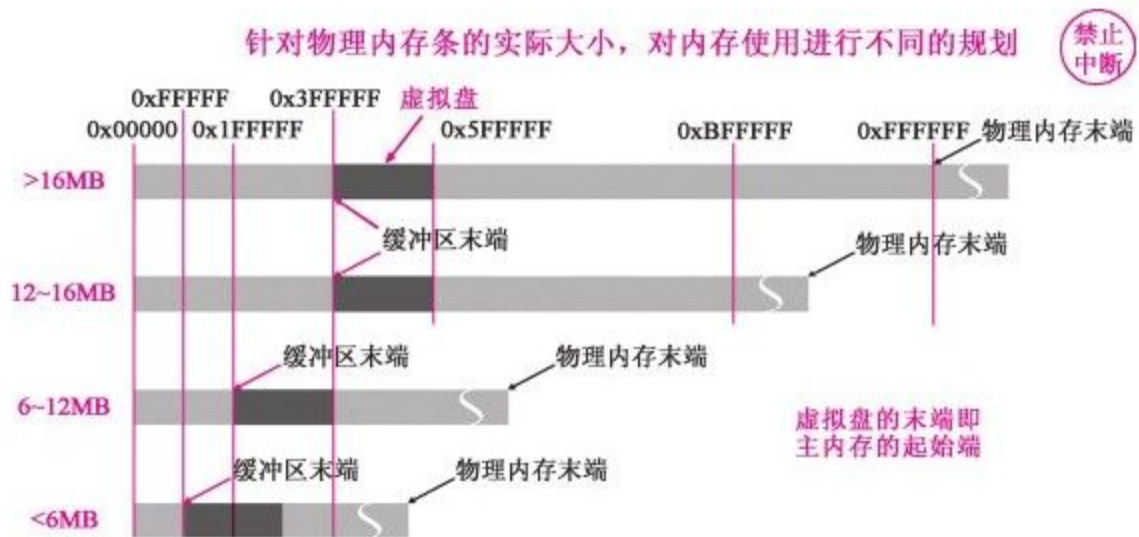


图 2-3 物理内存规划格局

调用`rd_init ()`函数，开始对虚拟盘进行设置，具体执行代码如下：

```
//代码路径：init/main.c:

void main (void)

{

.....

#ifdef RAMDISK

    main_memory_start+=rd_init
(main_memory_start,RAMDISK*1024) ;
```

```
#endif
```

```
.....
```

```
}
```

```
//代码路径: kernel/blk_drv/blk.h:
```

```
.....
```

```
#define NR_BLK_DEV 7
```

```
.....
```

```
struct blk_dev_struct{
```

```
void (*request_fn) (void) ;
```

```
struct request * current_request;
```

```
};
```

```
.....
```

```
#if (MAJOR_NR==1)
```

```
.....
```

```
#define DEVICE_REQUEST do_rd_request
```

```
.....
```

```
//代码路径: kernel/blk_drv/ll_rw_blk.c: //ll可以理解为low level
```

.....

```
struct blk_dev_struct blk_dev[NR_BLK_DEV]={  
  
    {NULL,NULL}, /*no_dev*/  
  
    {NULL,NULL}, /*dev mem*/  
  
    {NULL,NULL}, /*dev fd*/  
  
    {NULL,NULL}, /*dev hd*/  
  
    {NULL,NULL}, /*dev ttyx*/  
  
    {NULL,NULL}, /*dev tty*/  
  
    {NULL,NULL}/*dev lp*/  
  
};
```

.....

//代码路径: kernel/ramdisk.c:

.....

```
#define MAJOR_NR 1
```

.....

long rd_init (long mem_start,int length) //hd_init () 、floppy_init
() 与此类似

```

{

int

i;

char * cp;

    blk_dev[MAJOR_NR].request_fn=DEVICE_REQUEST; //挂接
do_rd_request ()

    rd_start= (char *) mem_start;

    rd_length=length;

    cp=rd_start;

    for (i=0; i<length; i++)

        *cp++='\0'; //初始化为0

    return (length) ;

}

```

在rd_init () 函数中，先要将虚拟盘区的请求项处理函数do_rd_request () 与图2-4中的请求项函数控制结构blk_dev[7]的第二项挂接。

blk_dev[7]的主要功能是将某一类设备与它对应的请求项处理函数挂钩。可以看出我们讨论的操作系统最多可以管理6类设备。请求项将在2.6节中详细解释。这个挂接动作意味着以后内核能够通过调用do_rd_request函数处理与虚拟盘相关的请求项操作。挂接之后，将虚拟盘所在的内存区域全部初始化为0。图2-4表示了rd_init（）函数的执行效果。

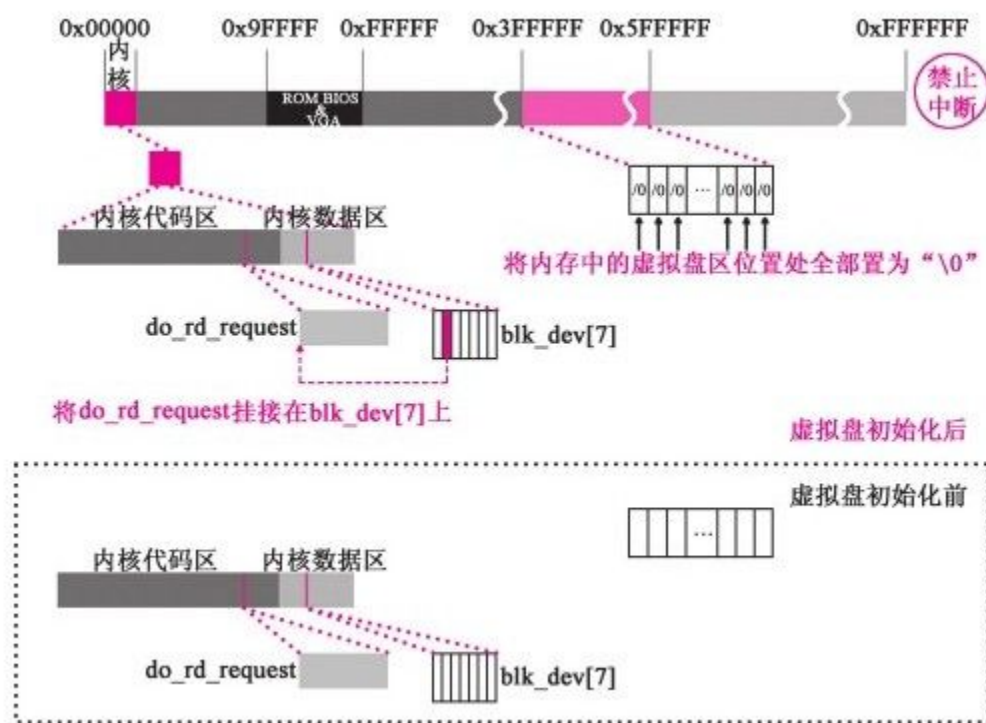


图 2-4 虚拟盘设置与初始化

最后将虚拟盘区的长度值返回。这个返回值将用来重新设置主内存区的起始位置。

2.4 内存管理结构mem_map初始化

对主内存区起始位置的重新确定，标志着主内存区和缓冲区的位置和大小已经全都确定了，于是系统开始调用mem_init（）函数。先对主内存区的管理结构进行设置，该过程如图2-5所示。

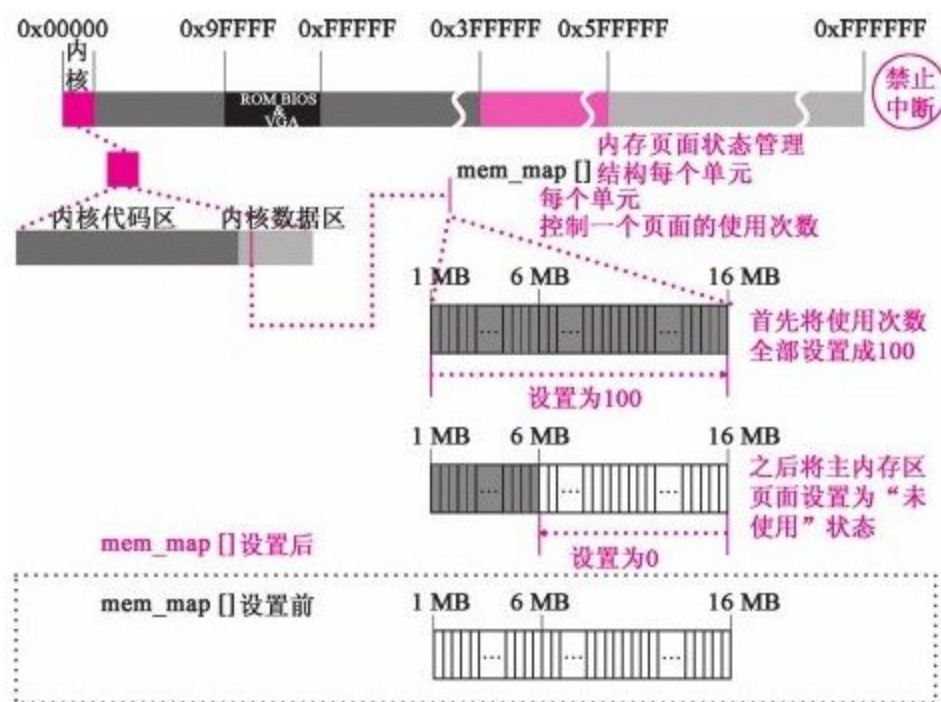


图 2-5 mem_map初始化

具体执行代码如下：

```
//代码路径： init/main.c:
```

```
void main (void)
```

```
{
```

```
.....
```

```
mem_init (main_memory_start,memory_end) ;
```

```
.....
```

```
}
```

```
//代码路径： mm/memory.c:
```

```
.....
```

```
#define LOW_MEM 0x100000//1 MB
```

```
#define PAGING_MEMORY (15*1024*1024)
```

```
#define PAGING_PAGES (PAGING_MEMORY >> 12) //15 MB的  
页数
```

```
#define MAP_NR (addr) ( ( (addr) -LOW_MEM) >> 12)
```

```
#define USED 100
```


.....

```
static long HIGH_MEMORY=0;
```

.....

```
static unsigned char mem_map[PAGING_PAGES]={0, };
```

.....

```
void mem_init (long start_mem,long end_mem)
```

```
{
```

```
int i;
```

```
HIGH_MEMORY=end_mem;
```

```
for (i=0; i<PAGING_PAGES; i++)
```

```
mem_map[i]=USED;
```

```
i=MAP_NR (start_mem) ; //start_mem为6 MB (虚拟盘之后)
```

```
end_mem-=start_mem;
```

```
end_mem>>=12; //16 MB的页数
```

```
while (end_mem-->0)
```

```
mem_map[i++]=0;
```

```
}
```

系统通过`mem_map[]`对1 MB以上的内存分页进行管理，记录一个页面的使用次数。

`mem_init ()` 函数先将所有的内存页面使用计数均设置成USED（100，即被使用），然后再将主内存中的所有页面使用计数全部清零，系统以后只把使用计数为0的页面视为空闲页面。

那么为什么系统对1 MB以内的内存空间不用这种分页方法管理呢？这是因为，操作系统的设计者对内核和用户进程采用了两套不同的分页管理方法。内核采用分页管理方法，线性地址和物理地址是完全一样的，是一一映射的，等价于内核可以直接获得物理地址。用户进程则不然，线性地址和物理地址差异很大，之间没有可递推的

逻辑关系。操作系统设计者的目的就是让用户进程无法通过线性地址推算出具体的物理地址，让内核能够访问用户进程，用户进程不能访问其他的用户进程，更不能访问内核。1 MB以内是内核代码和只有由内核管控的大部分数据所在内存空间，是绝对不允许用户进程访问的。1 MB以上，特别是主内存区主要是用户进程的代码、数据所在内存空间，所以采用专门用来管理用户进程的分页管理方法，这套方法当然不能用在内核上。详细内容请看第6章中的内存管理，深层次原因的分析请看第9章。

2.5 异常处理类中断服务程序挂接

不论是用户进程还是系统内核都要经常使用中断或遇到很多异常情况需要处理，如CPU在参与运算过程中，可能会遇到除零错误、溢出错误、边界检查错误、缺页错误.....免不了需要“异常处理”。中断技术也是广泛使用的，系统调用就是利用中断技术实现的。这些中断、异常都需要具体的服务程序来执行。`trap_init()` 函数将中断、异常处理的服务程序与IDT进行挂接，逐步重建中断服务体系，支持内核、进程在主机中的运算。挂接的具体过程及异常处理类中断服务程序在IDT中所占用的位置如图2-6所示。

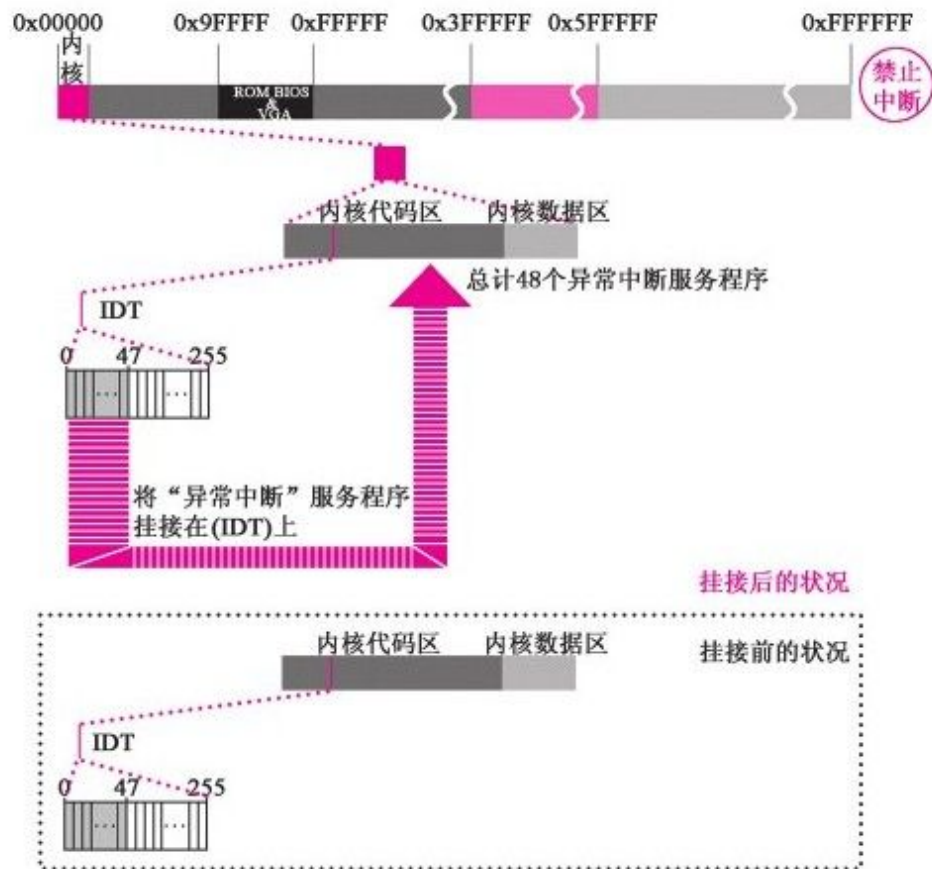


图 2-6 异常处理类中断服务程序挂接

执行代码如下：

//代码路径：init/main.c:

```
void main (void)
```

```
{
```

```
.....
```

```
trap_init () ;
```

```
.....
```

```
}
```

```
//代码路径: kernel/traps.c:
```

```
void trap_init (void)
```

```
{
```

```
int i;
```

```
set_trap_gate (0, &divide_error) ; //除零错误
```

```
set_trap_gate (1, &debug) ; //单步调试
```

```
set_trap_gate (2, &nmi) ; //不可屏蔽中断
```

```
set_system_gate (3, &int3) ; /*int3-5 can be called from all*/
```

```
set_system_gate (4, &overflow) ; //溢出
```

```
set_system_gate (5, &bounds) ; //边界检查错误
```

```
set_trap_gate (6, &invalid_op) ; //无效指令
```

```
set_trap_gate (7, &device_not_available) ; //无效设备
```

```
set_trap_gate (8, &double_fault) ; //双故障
```

```
set_trap_gate (9, &coprocessor_segment_overrun) ; //协处理器段  
越界
```

```

set_trap_gate (10, &invalid_TSS) ; //无效TSS

set_trap_gate (11, &segment_not_present) ; //段不存在

set_trap_gate (12, &stack_segment) ; //栈异常

set_trap_gate (13, &general_protection) ; //一般性保护异常

set_trap_gate (14, &page_fault) ; //缺页

set_trap_gate (15, &reserved) ; //保留

set_trap_gate (16, &coprocessor_error) ; //协处理器错误

for (i=17; i<48; i++) //都先挂接好，中断服务程序函数名初始
化为保留

set_trap_gate (i, &reserved) ;

set_trap_gate (45, &irq13) ; //协处理器

outb_p (inb_p (0x21) &0xfb, 0x21) ; //允许IRQ2中断请求

outb (inb_p (0xA1) &0xdf, 0xA1) ; //允许IRQ2中断请求

set_trap_gate (39, &parallel_interrupt) ; //并口

}

//代码路径: include\asm\system.h:

.....

#define_set_gate (gate_addr,type,dpl,addr) \

```

字 __asm__ ("movw%%dx, %%ax\n\t"//将edx的低字赋值给eax的低

"movw%0, %%dx\n\t"//%0对应第二个冒号后的第1行的"i"

"movl%%eax, %1\n\t"//%1对应第二个冒号后的第2行的"o"

"movl%%edx, %2"//%2对应第二个冒号后的第3行的"o"

: //这个冒号后面是输出，下面冒号后面是输入

: "i" ((short) (0x8000+ (dpl<<13) + (type<<8))), //立即数

"o" (* ((char *) (gate_addr))), //中断描述符前4个字节的地址

"o" (* (4+ (char *) (gate_addr))), //中断描述符后4个字节的地址

"d" ((char *) (addr)), "a" (0x00080000) // "d"对应edx, "a"对应eax

.....

#define set_trap_gate (n,addr) \

_set_gate (&idt[n], 15, 0, addr)

这些代码的目的就是要拼出第1章1.3.5节讲述过的中断描述符。为了便于阅读，复制在下面，如图2-7所示。



图 2-7 中断描述符

上述代码的执行效果如图2-8所示。

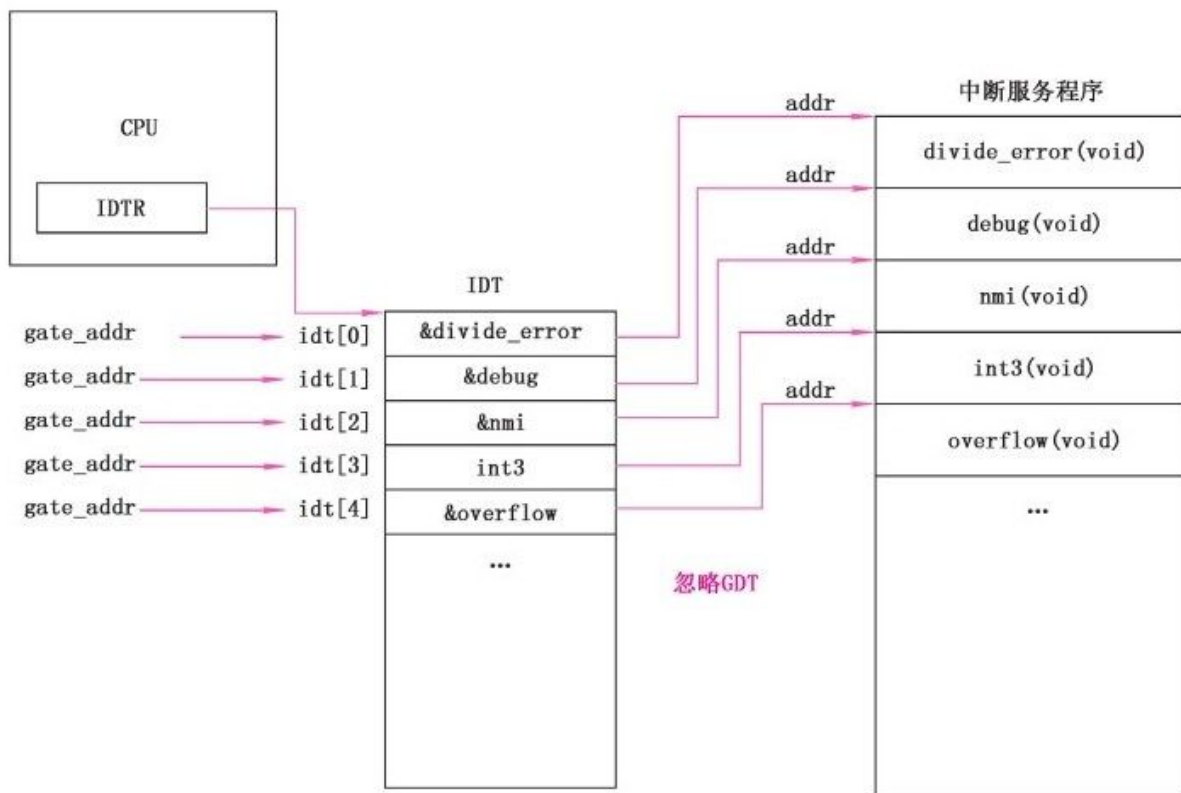


图 2-8 代码执行效果示意图

对比:

`set_trap_gate (0, ÷_error)`

`set_trap_gate (n,addr)`

`_set_gate (&idt[n], 15, 0, addr)`

`_set_gate (gate_addr,type,dpl,addr)`

可以看出，`n`是0；`gate_addr`是`&idt[0]`，也就是`idt`的第一项中断描述符的地址；`type`是15；`dpl`（描述符特权级）是0；`addr`是中断服务程序`divide_error (void)`的入口地址，如图2-9所示。

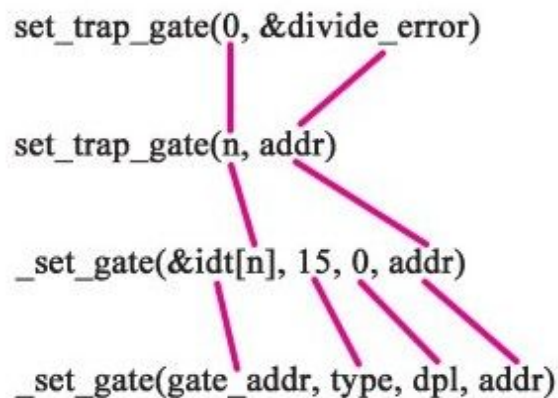


图 2-9 参数对应示意图

“`movw%%dx, %%ax\n\t`”是把`edx`的低字赋值给`eax`的低字；`edx`是`(char *) (addr)`，也就是`÷_error`；`eax`的值是`0x00080000`，这个数据

在head.s中就提到过，8应该看成1000，每一位都有意义，这样eax的值就是0x00080000+（（char*）（addr）的低字），其中的0x0008是段选择符，含义与第1章中讲解过的“jmp 0, 8”中的8一致。

"movw%0, %%dx\n\t"是把（short）（0x8000+（dpl<<13）+（type<<8））赋值给dx。别忘了，edx是（char*）（addr），也就是÷_error。

因为这部分数据是按位拼接的，必须计算精确，我们耐心详细计算一下：

0x8000就是二进制的1000 0000 0000 0000；

dpl是00, dpl < < 13就是000 0000 0000
0000;

type是15, type < < 8就是1111 0000 0000;

加起来就是1000 1111 0000 0000, 这就是dx的值。edx的计算结果就是 (char *) (addr) 的高字即÷_error的高字+1000 1111 0000 0000。

"movl%%eax, %1\n\t"是把eax的值赋给*
((char *) (gate_addr)), 就是赋给idt[0]的前
4字节。同理, "movl%%edx, %2"是把edx的值赋
给* (4+ (char *) (gate_addr)), 就是赋给
idt[0]的前后4字节。8字节合起来就是完整的
idt[0]。拼接的效果如图2-10所示。

IDT中的第一项除零错误中断描述符初始化完毕，其余异常处理服务程序的中断描述符初始化过程大同小异。后续介绍的所有中断服务程序与IDT的初始化基本上都是以这种方式进行的。

`set_system_gate (n,addr)` 与 `set_trap_gate (n,addr)` 用的 `_set_gate (gate_addr,type,dpl,addr)` 是一样的；差别是 `set_trap_gate` 的 `dpl` 是 0，而 `set_system_gate` 的 `dpl` 是 3。 `dpl` 为 0 的意思是只能由内核处理， `dpl` 为 3 的意思是系统调用可以由 3 特权级（也就是用户特权级）调用。

有关特权级更深入的内容，请参看《Intel IA-32 Architectures Software Developer's Manual Volume 3.pdf》（可以到Intel官方网站下载）。

接下来将IDT的int 0x11～int 0x2F都初始化，将IDT中对应的指向中断服务程序的指针设置为reserved（保留）。

设置协处理器的IDT项。

允许主8259A中断控制器的IRQ2、IRQ3的中断请求。

设置并口（可以接打印机）的IDT项。

32位中断服务体系是为适应“被动响应”中断信号机制而建立的。其特点、技术路线是这样的：一方面，硬件产生信号传达给8259A，8259A对信号进行初步处理并视CPU执行情况传递中断信号给CPU；另一方面，CPU如果没有接收到信号，就不断地处理正在执行的程序，如果接收到

信号，就打断正在执行的程序并通过IDT找到具体的中断服务程序，让其执行，执行完后，返回刚才打断的程序点继续执行。如果又接收到中断信号，就再次处理中断.....

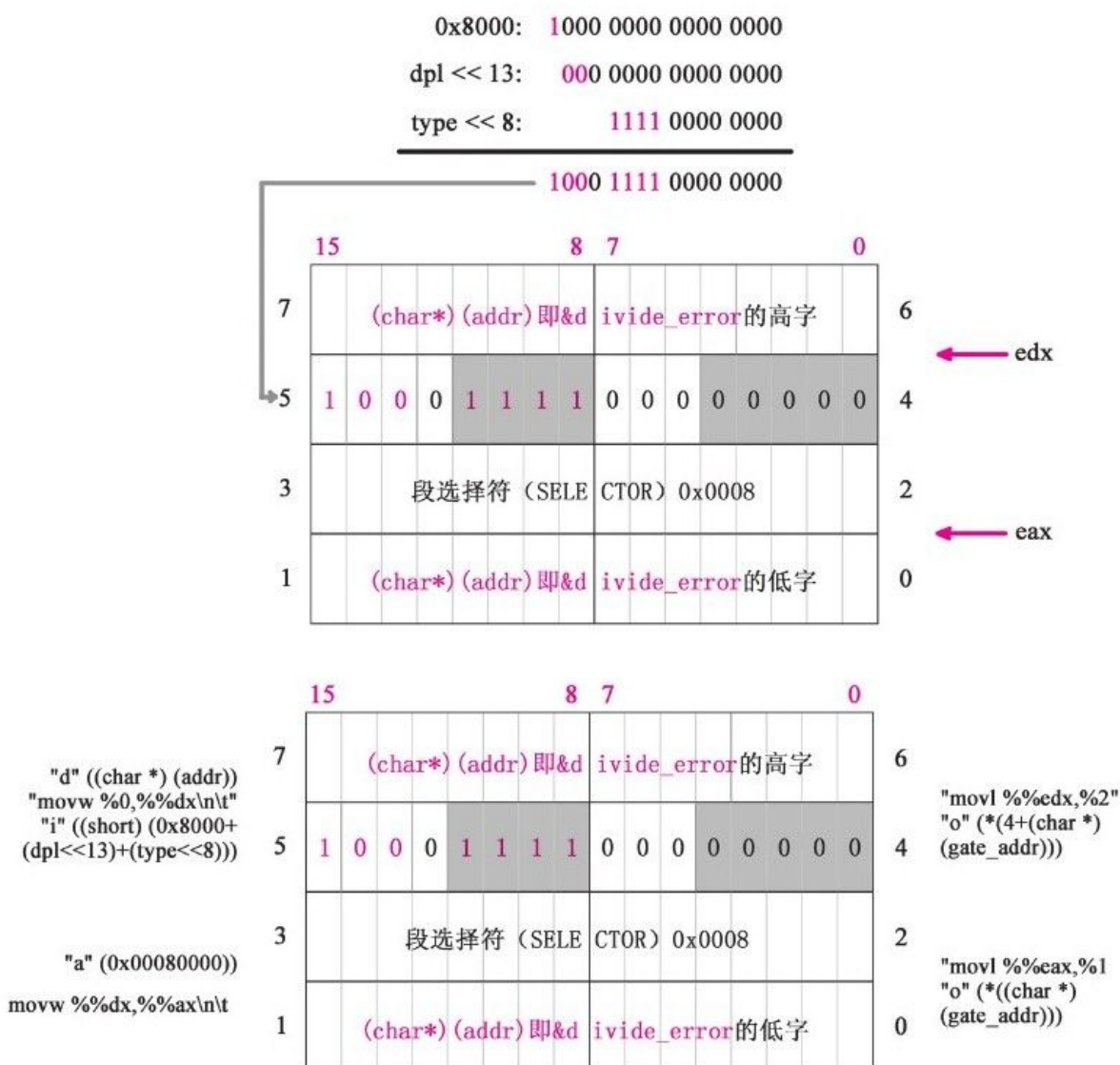


图 2-10 拼接效果

最原始的设计不是这样，那时候CPU每隔一段时间就要对所有硬件进行轮询，以检测它的工作是否完成，如果没有完成就继续轮询，这样就消耗了CPU处理用户程序的时间，降低了系统的综合效率。可见，CPU以“主动轮询”的方式来处理信号是非常不划算的。以“被动响应”模式替代“主动轮询”模式来处理主机与外设的I/O问题，是计算机历史上的一大进步。

2.6 初始化块设备请求项结构

Linux 0.11将外设分为两类：一类是块设备，另一类是字符设备。块设备将存储空间等分为若干同样大小的称为块的小存储空间，每个块有块号，可以独立、随机读写。硬盘、软盘都是块设备。字符设备以字符为单位进行I/O通信。键盘、早期黑屏命令行显示器都是字符设备。

进程要想与块设备进行沟通，必须经过主机内存中的缓冲区。请求项管理结构request[32]就是操作系统管理缓冲区中的缓冲块与块设备上逻辑块之间读写关系的数据结构。

请求项在进程与块设备进行I/O通信的总体关系如图2-11所示。

操作系统根据所有进程读写任务的轻重缓急，决定缓冲块与块设备之间的读写操作，并把需要操作的缓冲块记录在请求项上，得到读写块设备操作指令后，只根据请求项中的记录来决定当前需要处理哪个设备的哪个逻辑块。

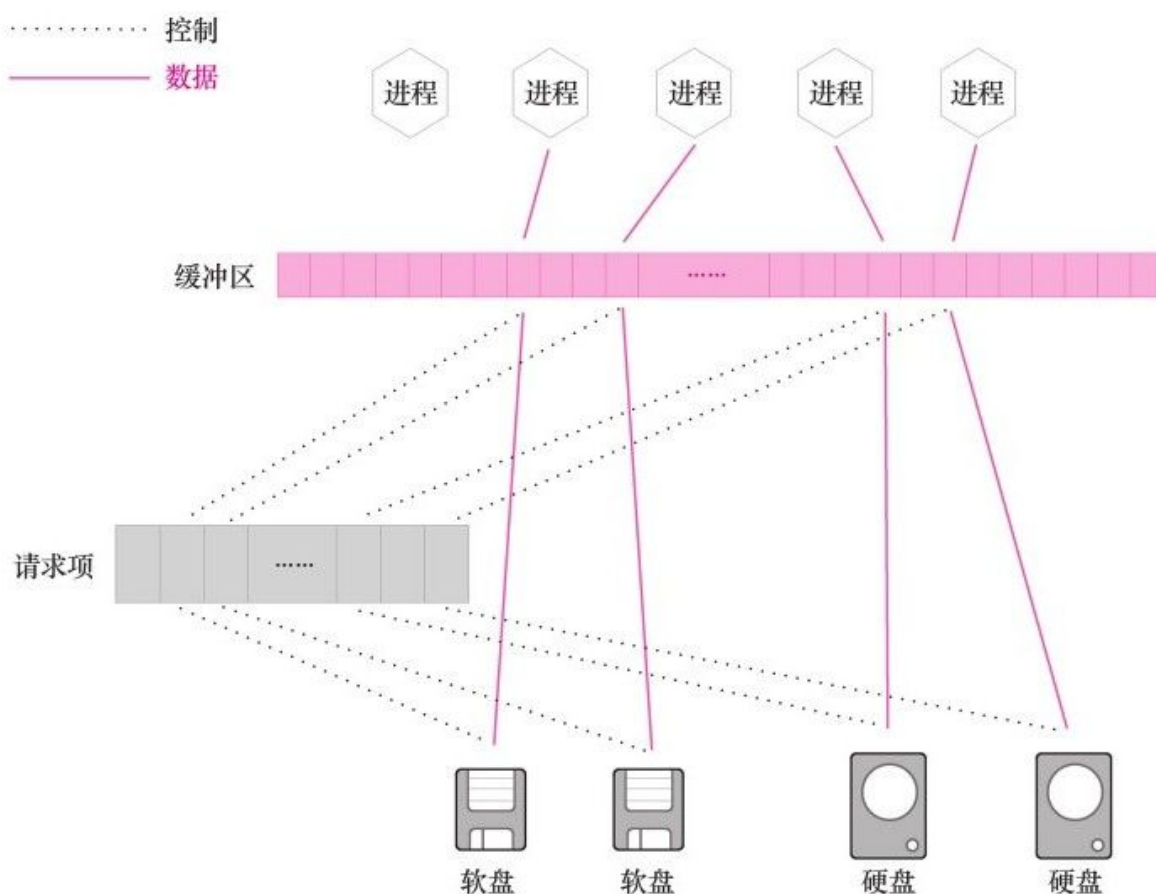


图 2-11 总体关系示意图

执行代码如下：

```
//代码路径: init/main.c:
```

```
void main (void)
```

```
{
```

```
.....
```

```
blk_dev_init () ;
```

```
.....
```

```
}
```

```
//代码路径: kernel/blk_dev/blk.h:
```

```
.....
```

```
#define NR_REQUEST 32
```

```
struct request{
```

```
int dev; /*-1 if no request*/
```

```
int cmd; /*READ or WRITE*/
```

```
int errors;
```

```
unsigned long sector;
```

```

unsigned long nr_sectors;

char * buffer;

struct task_struct * waiting;

struct buffer_head * bh;

struct request * next; //说明request可以构成链表

};

.....

//代码路径: kernel/blk_dev/ll_rw_block.c:

.....

struct request request[NR_REQUEST]; //数组链表

.....

void blk_dev_init (void)

{

int i;

for (i=0; i<NR_REQUEST; i++) {

request[i].dev=-1; //设置为空闲

request[i].next=NULL; //互不挂接

```

```
}  
  
}
```

注意： request[32]是一个由数组构成的链表；
request[i].dev=-1说明了这个请求项还没有具体对
应哪个设备，这个标志将来会被用来判断对应该
请求项的当前设备是否空闲；
request[i].next=NULL说明这时还没有形成请求项
队列。初始化的过程和效果如图2-12所示。

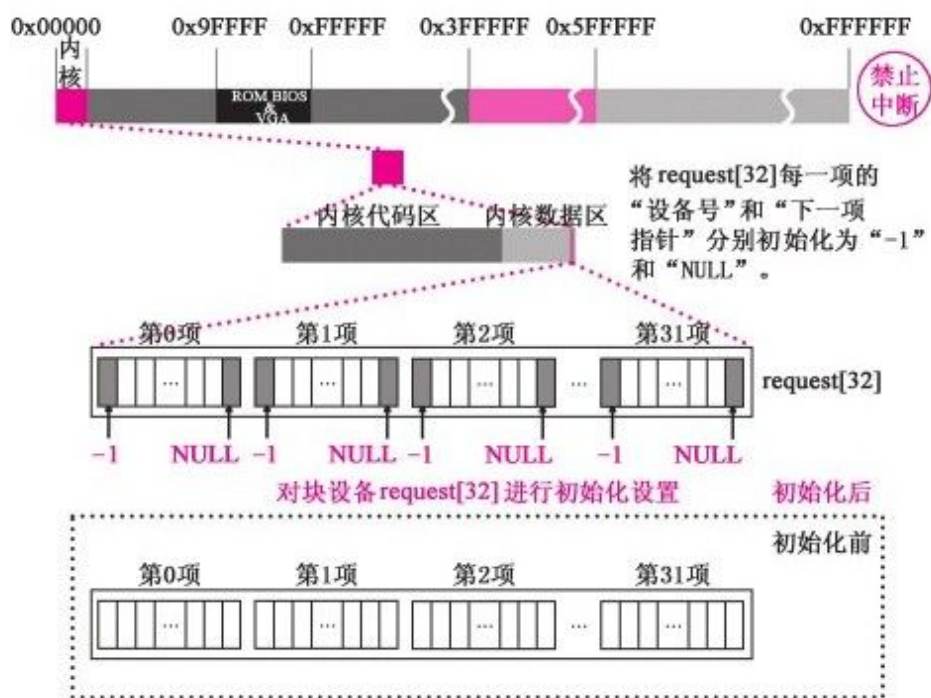


图 2-12 初始化块设备请求项结构

2.7 与建立人机交互界面相关的外设的中断服务程序挂接

Linux在操作系统源代码中本来设计了chr_dev_init () 函数，明显是要用这个函数初始化字符设备，但我们可以看到这是一个空函数。Linux又设计了tty_init () 函数，内容就是初始化字符设备。有人解释tty是teletype。

字符设备的初始化为进程与串行口（可以通信、连接鼠标.....）、显示器以及键盘进行I/O通信准备工作环境，主要是对串行口、显示器、键盘进行初始化设置，以及与此相关的中断服务程序与IDT挂接。在tty_init () 函数中，先调用

rs_init () 函数来设置串行口，再调用con_init
() 函数来设置显示器，具体执行代码如下：

//代码路径: init/main.c:

```
void main (void)
```

```
{
```

```
.....
```

```
tty_init () ;
```

```
.....
```

```
}
```

//代码路径: kernel/chr_dev/tty_io.c:

```
void tty_init (void)
```

```
{
```

```
rs_init () ;
```

```
con_init ()
```

```
}
```

2.7.1 对串行口进行设置

把两个串行口中断服务程序与IDT相挂接，然后根据tty_table数据结构中的内容对这两个串行口进行初始化设置，包括设置线路控制寄存器的DLAB位、设置发送的波特率因子、设置DTR和RTS.....最后，允许主8259A芯片的IRQ3和IRQ4发送中断请求。

挂接的具体过程及挂接后的效果如图2-13所示。

执行代码如下：

```
//代码路径: kernel/chr_dev/serial.c:
```

```
void rs_init (void)
```

```
{
```

```
2.5    set_intr_gate (0x24, rs1_interrupt) ; //设置串行口1中断, 参看  
  
        set_intr_gate (0x23, rs2_interrupt) ; //设置串行口2中断  
  
        init (tty_table[1].read_q.data) ; //初始化串行口1  
  
        init (tty_table[2].read_q.data) ; //初始化串行口2  
  
        outb (inb_p (0x21) & 0xE7, 0x21) ; //允许IRQ3, IRQ4  
  
    }
```

两个串行口中断处理程序与IDT的挂接函数 `set_intr_gate ()` 与2.5中介绍过的 `set_trap_gate ()` 函数类似, 可参看前面对 `set_trap_gate ()` 函数的讲解。它们的差别是 `set_trap_gate ()` 函数的 `type` 是15 (二进制的1111), 而 `set_intr_gate ()` 的 `type` 是14 (二进制的1110)。

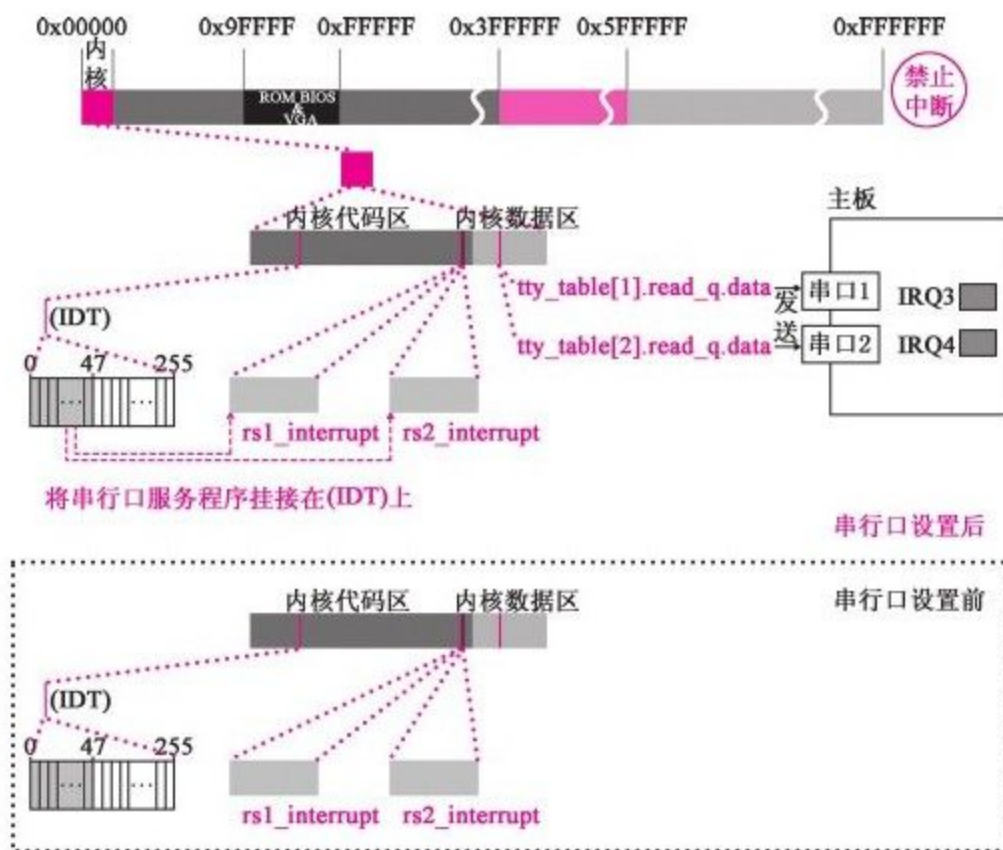


图 2-13 串行口中断服务程序挂接

2.7.2 对显示器进行设置

根据机器系统数据提供的显卡是“单色”还是“彩色”来设置配套信息。由于在Linux 0.11那个时代，大部分显卡器是单色的，所以我们假设显卡的属性是单色EGA。那么显存的位置就要被设置为0xb0000~0xb8000，索引寄存器端口被设置为0x3b4，数据寄存器端口被设置为0x3b5，再将显卡的属性——EGA这三个字符，显示在屏幕上。另外，再初始化一些用于滚屏的变量，其中包括滚屏的起始显存地址、滚屏结束显存地址、最顶端行号以及最低端行号。效果如图2-14所示。

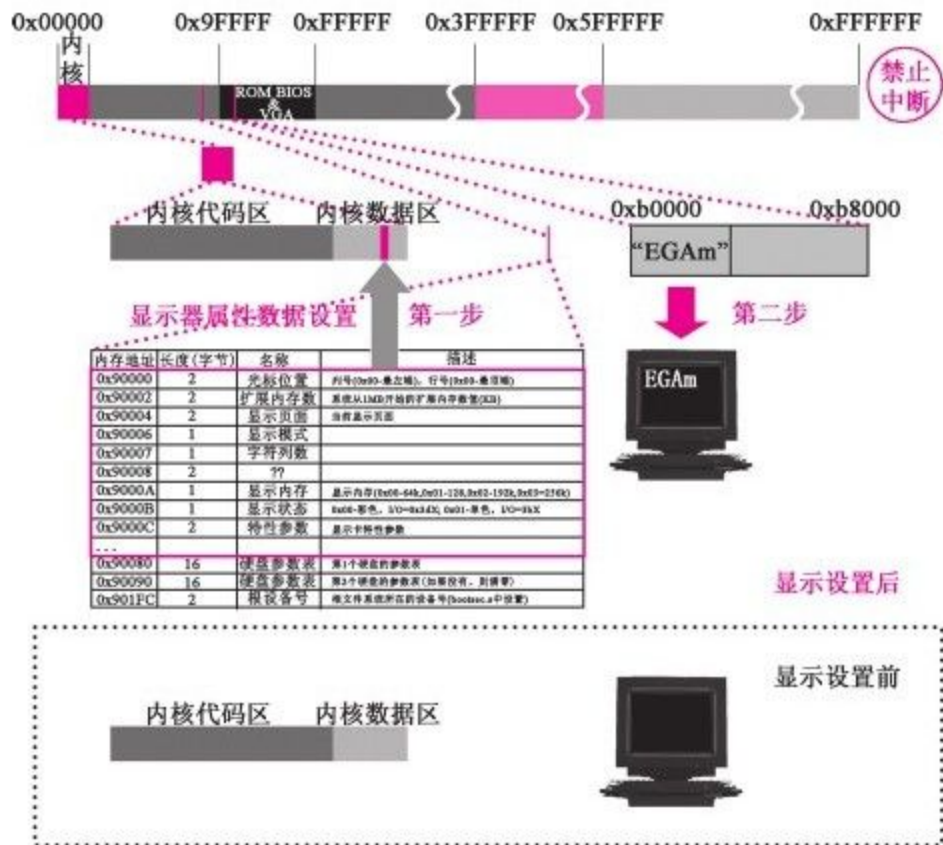


图 2-14 显示器设置

2.7.3 对键盘进行设置

对键盘进行设置是先将键盘中断服务程序与IDT相挂接，然后取消8259A中对键盘中断的屏蔽，允许IRQ1发送中断信号，通过先禁止键盘工作、再允许键盘工作，键盘便能够使用了。键盘中断处理程序与IDT的挂接函数set_intr_gate（）与前面讲解过的set_trap_gate（）函数类似，参看对set_trap_gate（）函数的讲解。

效果如图2-15所示。

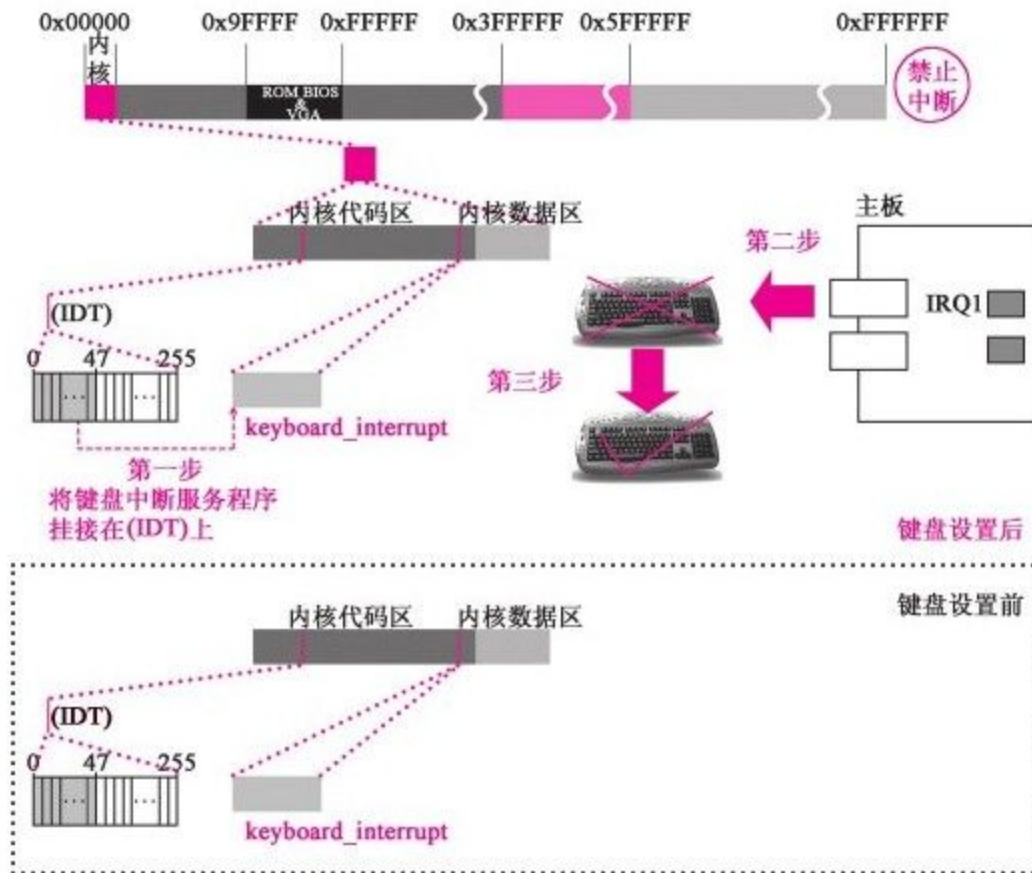


图 2-15 键盘设置

执行代码如下：

```
//代码路径： kernel/chr_dev/console.c:
```

```
.....
```

```
#define ORIG_X (* (unsigned char *) 0x90000)
```

```
#define ORIG_Y (* (unsigned char *) 0x90001)
```



```

#define ORIG_VIDEO_PAGE (* (unsigned short *) 0x90004)

#define ORIG_VIDEO_MODE ( (* (unsigned short *) 0x90006)
& 0xff)

#define ORIG_VIDEO_COLS ( ( (* (unsigned short *)
0x90006) & 0xff00) > > 8)

#define ORIG_VIDEO_LINES (25)

#define ORIG_VIDEO_EGA_AX (* (unsigned short *) 0x90008)

#define ORIG_VIDEO_EGA_BX (* (unsigned short *) 0x9000a)

#define ORIG_VIDEO_EGA_CX (* (unsigned short *) 0x9000c)

#define VIDEO_TYPE_MDA 0x10/*Monochrome Text Display*/

#define VIDEO_TYPE_CGA 0x11/*CGA Display*/

#define VIDEO_TYPE_EGAM 0x20/*EGA/VGA in Monochrome
Mode*/

#define VIDEO_TYPE_EGAC 0x21/*EGA/VGA in Color Mode*/

#define NPAR 16

.....

void con_init (void)

{

register unsigned char a;

```

```

char * display_desc="????";

char * display_ptr;

video _num_columns=ORIG_VIDEO_COLS; //参看机器系统数据

video _size_row=video_num_columns*2;

video _num_lines=ORIG_VIDEO_LINES;

video _page=ORIG_VIDEO_PAGE; //参看机器系统数据

video _erase_char=0x0720;

if (ORIG_VIDEO_MODE==7) /*Is this a monochrome display?*/

{

video _mem_start=0xb0000;

video _port_reg=0x3b4;

video _port_val=0x3b5;

if ( (ORIG_VIDEO_EGA_BX&0xff) !=0x10) //参看机器系统
数据

{

video _type=VIDEO_TYPE_EGAM;

video _mem_end=0xb8000;

```

```

display _desc="EGAm";

}

else

{

video _type=VIDEO_TYPE_MDA;

video _mem_end=0xb2000;

display _desc="*MDA";

}

}

else/*If not,it is color.*/

{

video _mem_start=0xb8000;

video _port_reg=0x3d4;

video _port_val=0x3d5;

if ( (ORIG_VIDEO_EGA_BX&0xff) !=0x10) //参看机器系统
数据
{

```

```

video _type=VIDEO_TYPE_EGAC;

video _mem_end=0xbc000;

display _desc="EGAc";

}

else

{

video _type=VIDEO_TYPE_CGA;

video _mem_end=0xba000;

display _desc="*CGA";

}

}

/*Let the user known what kind of display driver we are using*/

display _ptr= ( (char *) video_mem_start) +video_size_row-8;

while (*display_desc)

{

*display_ptr++=*display_desc++;

display _ptr++;

```

```

}

/*Initialize the variables used for scrolling (mostly EGA/VGA) */

origin=video_mem_start;

scr_end=video_mem_start+video_num_lines * video_size_row;

top=0;

bottom=video_num_lines;

gotoxy (ORIG_X,ORIG_Y) ; //参看机器系统数据

set_trap_gate (0x21, &keyboard_interrupt) ; //设置键盘中断,
参看2.5节

outb_p (inb_p (0x21) & 0xfd, 0x21) ; //取消对键盘中断的屏蔽, 允许IRQ1

a=inb_p (0x61) ;

outb_p (a|0x80, 0x61) ; //禁止键盘工作

outb (a, 0x61) ; //再允许键盘工作

}

```

2.8 开机启动时间设置

开机启动时间是大部分与时间相关的计算的基础。操作系统中一些程序的运算需要时间参数；很多事务的处理也都要用到时间，比如文件修改的时间、文件最近访问的时间、i节点自身的修改时间等。有了开机启动时间，其他时间就可据此推算出来。

具体执行步骤是：CMOS是主板上的一个小存储芯片，系统通过调用time_init（）函数，先对它上面记录的时间数据进行采集，提取不同等级的时间要素，比如秒（time.tm_sec）、分（time.tm_min）、年（time.tm_year）等，然后对

这些要素进行整合，并最终得出开机启动时间
(startup_time) 。

执行代码如下：

```
//代码路径: init/main.c:

void main (void)

{

.....

time _init () ;

.....

}

#define CMOS_READ (addr) ({V/读CMOS实时时钟信息

outb _p (0x80|addr, 0x70) ; V/0x80|addr读CMOS地址, 0x70写
端口

inb _p (0x71) ; V/0x71读端口

})
```

```

#define BCD_TO_BIN (val) ( (val) = ( (val) &15) +
( (val) >>4) *10) //十进制转二进制

static void time_init (void)

{

struct tm time;

do{

time.tm_sec=CMOS_READ (0) ; //当前时间的秒值， 以下类推

time.tm_min=CMOS_READ (2) ;

time.tm_hour=CMOS_READ (4) ;

time.tm_mday=CMOS_READ (7) ;

time.tm_mon=CMOS_READ (8) ;

time.tm_year=CMOS_READ (9) ;

}while (time.tm_sec !=CMOS_READ (0) ) ;

BCD_TO_BIN (time.tm_sec) ;

BCD_TO_BIN (time.tm_min) ;

BCD_TO_BIN (time.tm_hour) ;

BCD_TO_BIN (time.tm_mday) ;

```



```
BCD_TO_BIN (time.tm_mon) ;
```

```
BCD_TO_BIN (time.tm_year) ;
```

```
time.tm_mon--;
```

```
startup_time=kernel_mkttime (&time) ; //开机时间，从1970年1月1日0时计算
```

```
}
```

```
//代码路径: include\asm\io.h: //嵌入汇编参看trap_init的注释
```

```
#define outb_p (value,port) \//将value写到port
```

```
__asm__ ("outb%%al, %%dx\n"
```

```
"\tjmp 1f\n"//jmp到下面的第一个1: 处，目的是延迟
```

```
"1: \tjmp 1f\n"
```

```
"1: ": "a" (value) , "d" (port) )
```

```
#define inb_p (port) ({\
```

```
unsigned char_v; \
```

```
__asm__volatile ("inb%%dx, %%al\n"//volatile, 禁止编译器优化  
下列代码
```

```
"\tjmp 1f\n"//延迟
```

```
"1: \tjmp 1f\n"
```

```
"1: ": "=a" (_v) : "d" (port) ) ; \
_v; \
})
```

计算过程及开机启动时间在内存中的存储位置如图2-16所示。

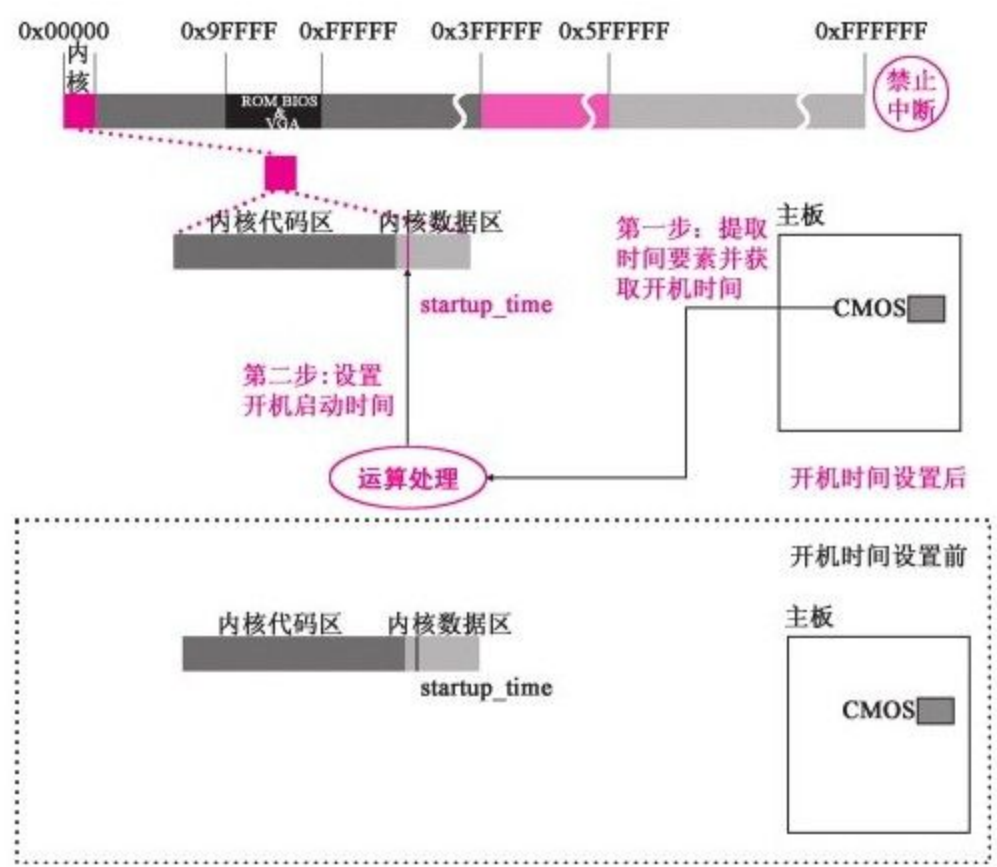


图 2-16 开机启动时间设置

2.9 初始化进程0

进程0是Linux操作系统中运行的第一个进程，也是Linux操作系统父子进程创建机制的第一个父进程。下面讲解的内容对进程0能够在主机中正常运算的影响最为重要和深远，主要包含如下三方面的内容。

1) 系统先初始化进程0。进程0管理结构task_struct的母本（init_task={INIT_TASK, }）已经在代码设计阶段事先设计好了，但这并不代表进程0已经可用了，还要将进程0的task_struct中的LDT、TSS与GDT相挂接，并对GDT、task[64]以及与进程调度相关的寄存器进行初始化设置。

2) Linux 0.11作为一个现代操作系统，其最重要的标志就是能够支持多进程轮流执行，这要求进程具备参与多进程轮询的能力。系统这里对时钟中断进行设置，以便在进程0运行后，为进程0以及后续由它直接、间接创建出来的进程能够参与轮转奠定基础。

3) 进程0要具备处理系统调用的能力。每个进程在运算时都可能需要与内核进行交互，而交互的端口就是系统调用程序。系统通过函数 `set_system_gate` 将 `system_call` 与 IDT 相挂接，这样进程0就具备了处理系统调用的能力了。这个 `system_call` 就是系统调用的总入口。

进程0只有具备了以上三种能力才能保证将来在主机中正常地运行，并将这些能力遗传给后续

建立的进程。

这三点的实现都是在`sched_init ()` 函数中实现的，具体代码如下：

//代码路径: `init/main.c`:

```
void main (void)
```

```
{
```

```
.....
```

```
    sched_init ();
```

```
.....
```

```
}
```

//代码路径: `kernel/sched.c`:

```
.....
```

```
#define LATCH (1193180/HZ) //每个时间片的振荡次数
```

```
.....
```

```
union task_union{//task_struct与内核栈的共用体
```

```
struct task_struct task;
```

```
char stack[PAGE_SIZE]; //PAGE_SIZE是4 KB
```

```
};
```

```
static union task_union init_task={INIT_TASK, }; //进程0的  
task_struct
```

```
.....
```

```
//初始化进程槽task[NR_TASKS]的第一项为进程0，即task[0]为进  
程0占用
```

```
struct task_struct * task[NR_TASKS]={& (init_task.task) , };
```

```
.....
```

```
void sched_init (void)
```

```
{
```

```
int i;
```

```
struct desc_struct * p;
```

```
if (sizeof (struct sigaction) !=16)
```

```
panic ("Struct sigaction MUST be 16 bytes" ) ;
```

```
set_tss_desc (gdt+FIRST_TSS_ENTRY, &  
(init_task.task.tss) ) ; //设置TSS0
```

```
    set_ldt_desc (gdt+FIRST_LDT_ENTRY, &
(init_task.task.ldt) ) ; //设置LDT0
```

p=gdt+2+FIRST_TSS_ENTRY; //从GDT的6项，即TSS1开始向上全部清零，并且将进程槽从

for (i=1; i<NR_TASKS; i++) { //1往后的项清空。0项为进程0所用

```
task[i]=NULL;
```

```
p->a=p->b=0;
```

```
p++;
```

```
p->a=p->b=0;
```

```
p++;
```

```
}
```

```
/*Clear NT,so that we won't have troubles with that later on*/
```

```
__asm__ ("pushfl; andl$0xffffbfff, (%esp) ; popfl" ) ;
```

```
ltr (0) ; //重要！将TSS挂接到TR寄存器
```

```
lldt (0) ; //重要！将LDT挂接到LDTR寄存器
```

outb _p (0x36, 0x43) ; /*binary,mode 3, LSB/MSB,ch 0*///设置定时器

```
outb _p (LATCH&0xff, 0x40) ; /*LSB*///每10毫秒一次时钟中断
```

```

    outb (LATCH >> 8, 0x40) ; /*MSB*/

    set_intr_gate (0x20, &timer_interrupt) ; //重要！设置时钟中
断，进程调度的基础

    outb (inb_p (0x21) & ~0x01, 0x21) ; //允许时钟中断

    set_system_gate (0x80, &system_call) ; //重要！设置系统调用
总入口

}

//代码路径: include/linux/sched.h: ////嵌入汇编参看trap_init的注释
.....

#define FIRST_TSS_ENTRY 4//参看图2-15中GDT的4项，即TSS0入
口

#define FIRST_LDT_ENTRY (FIRST_TSS_ENTRY+1) //同上，5
项即LDT0入口

#define _TSS (n) ( ( ( (unsigned long) n) << 4) +
(FIRST_TSS_ENTRY<<3) )

#define _LDT (n) ( ( ( (unsigned long) n) << 4) +
(FIRST_LDT_ENTRY<<3) )

#define ltr (n) __asm__ ("ltr%%ax": "a" (_TSS (n)) )

#define lldt (n) __asm__ ("lldt%%ax": "a" (_LDT (n)) )

.....

```


//代码路径: include\asm\system.h:

.....

```
#define set_intr_gate (n,addr) \  
_set_gate (&idt[n], 14, 0, addr)
```

```
#define set_trap_gate (n,addr) \  
_set_gate (&idt[n], 15, 0, addr)
```

```
#define set_system_gate (n,addr) \  
_set_gate (&idt[n], 15, 3, addr)
```

.....

#define set_tssldt_desc (n,addr,type) //嵌入汇编参看trap_init的注释

__asm__ ("movw\$104, %1\n\t"//将104, 即1101000存入描述符的第1、2字节

"movw%%ax, %2\n\t"//将tss或ldt基地址的低16位存入描述符的第3、4字节

"rorl\$16, %%eax\n\t"//循环右移16位, 即高、低字互换

"movb%%al, %3\n\t"//将互换完的第1字节, 即地址的第3字节存入第5字节

"movb\$"type", %4\n\t"//将0x89或0x82存入第6字节

"movb\$0x00, %5\n\t"\\将0x00存入第7字节

"movb%%ah, %6\n\t"\\将互换完的第2字节，即地址的第4字节存入第8字节

"rorl\$16, %%eax"\\复原eax

: "a" (addr) , "m" (* (n)) , "m" (* (n+2)) , "m" (* (n+4)) , \

"m" (* (n+5)) , "m" (* (n+6)) , "m" (* (n+7)) \

// "m" (* (n)) 是gdt第n项描述符的地址开始的内存单元

// "m" (* (n+2)) 是gdt第n项描述符的地址向上第3字节开始的内存单元

//其余依此类推

)

//n: gdt的项值, addr: tss或ldt的地址, 0x89对应tss, 0x82对应ldt

#define set_tss_desc (n,addr) _set_tssldt_desc (((char *) (n)) , addr, "0x89")

#define set_ldt_desc (n,addr) _set_tssldt_desc (((char *) (n)) , addr, "0x82")

//代码路径: include/linux/sched.h:

.....

struct tss_struct{

```
long back_link; /*16 high bits zero*/
```

```
long esp0;
```

```
long ss0; /*16 high bits zero*/
```

```
long esp1;
```

```
long ss1; /*16 high bits zero*/
```

```
long esp2;
```

```
long ss2; /*16 high bits zero*/
```

```
long cr3;
```

```
long eip;
```

```
long eflags;
```

```
long eax,ecx,edx,ebx;
```

```
long esp;
```

```
long ebp;
```

```
long esi;
```

```
long edi;
```

```
long es; /*16 high bits zero*/
```

```
long cs; /*16 high bits zero*/
```

```
long ss; /*16 high bits zero*/

long ds; /*16 high bits zero*/

long fs; /*16 high bits zero*/

long gs; /*16 high bits zero*/

long ldt; /*16 high bits zero*/

long trace_bitmap; /*bits: trace 0, bitmap 16-31*/

struct i387_struct i387;

};

struct task_struct{

/*these are hardcoded-don't touch*/

long state; /*-1 unrunnable, 0 runnable, > 0 stopped*/

long counter;

long priority;

long signal;

struct sigaction sigaction[32];

long blocked; /*bitmap of masked signals*/

/*various fields*/
```

```
int exit_code;

unsigned long start_code,end_code,end_data,brk,start_stack;

long pid,father,pgrp,session,leader;

unsigned short uid,euid,suid;

unsigned short gid,egid,sgid;

long alarm;

long utime,stime,cutime,cstime,start_time;

unsigned short used_math;

/*file system info*/

int tty; /*-1 if no tty,so it must be signed*/

unsigned short umask;

struct m_inode * pwd;

struct m_inode * root;

struct m_inode * executable;

unsigned long close_on_exec;

struct file * filp[NR_OPEN];

/*ldt for this task 0-zero 1-cs 2-ds&ss*/
```

```

struct desc_struct ldt[3];

/*tss for this task*/

struct tss_struct tss;

};

/*进程0的task_struct

*INIT_TASK is used to set up the first task table,touch at

*your own risk! .Base=0, limit=0x9ffff (=640kB)

*/

#define INIT_TASK\

/*state etc*/{0, 15, 15, V/就绪态, 15个时间片

/*signals*/0, {}, }, 0, \

/*ec,brk.....*/0, 0, 0, 0, 0, 0, \

/*pid etc..*/0, -1, 0, 0, 0, V/进程号0

/*uid etc*/0, 0, 0, 0, 0, 0, \

/*alarm*/0, 0, 0, 0, 0, 0, \

/*math*/0, \

/*fs info*/-1, 0022, NULL,NULL,NULL, 0, \

```

```

/*filp*/{NULL, }, \

{\

{0, 0}, \

/*ldt*/{0x9f, 0xc0fa00}, \

{0x9f, 0xc0f200}, \

}, \

/*tss*/{0, PAGE_SIZE+ (long) &init_task, 0x10, 0, 0, 0, 0,
(long) &pg_dir, \

0, 0, 0, 0, 0, 0, 0, 0, \efags的值, 决定了cli这类指令只能在0特权级使用

0, 0, 0x17, 0x17, 0x17, 0x17, 0x17, 0x17, \

_LDT (0) , 0x80000000, \

{} \

}, \

}

```

2.9.1 初始化进程0

sched_init函数比较难理解的是下面两行:

```
    set_tss_desc (gdt+FIRST_TSS_ENTRY, &
(init_task.task.tss) ) ;
```

```
    set_ldt_desc (gdt+FIRST_LDT_ENTRY, &
(init_task.task.ldt) ) ;
```

这两行代码的目的就是要像图2-17表现的那样在GDT中初始化进程0所占的4、5两项，即初始化TSS0和LDT0。

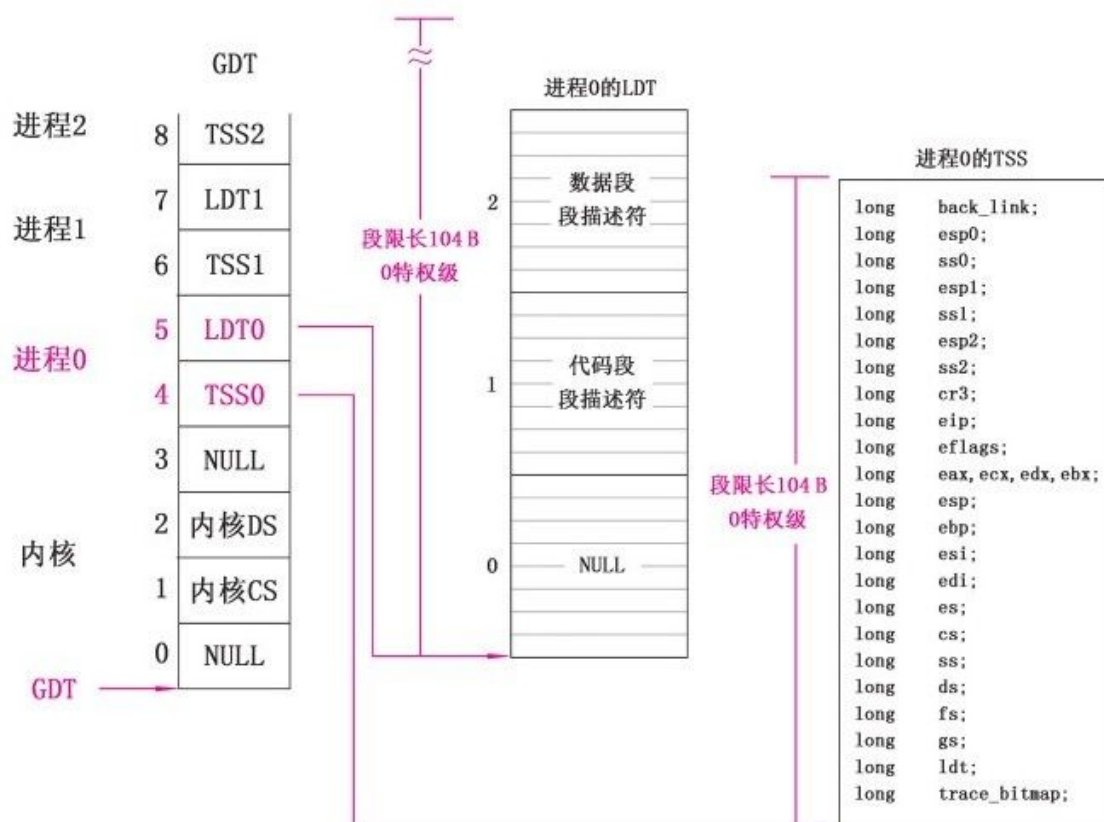


图 2-17 GDT、LDT、TSS关系示意图

另外，要拼出图2-18所示的结构。我们以TSS0为例，参看源代码中的注释，可以绘出图2-19。LDT0类似。



图 2-18 段描述符结构图

对比源代码、注释和图，可以看出，
`movw$104, %1`是将104赋给了段限长15: 0的部分；粒度G为0，说明限长就是104字节，而TSS除去`struct i387_struct i387`后长度正好是104字节。
LDT是 $3 \times 8 = 24$ 字节，所以104字节限长够用。TSS的类型是0x89，即二进制的10001001，可以看出`movb$"type", %4`在给type赋值1001的同时，顺便

将P、DPL、S字段都赋值好了。同理，
movb\$0x00, %5在给段限长19: 16部分赋值0000
的同时，顺便将G、D/B、保留、AVL字段都赋值
好了。



图 2-19 TSS0结构图

进程0的task_struct是由操作系统设计者事先写好的，就是sched.h中的INIT_TASK（参看上面相关源代码和注释，其结构示意见图2-20），并用INIT_TASK的指针初始化task[64]的0项。

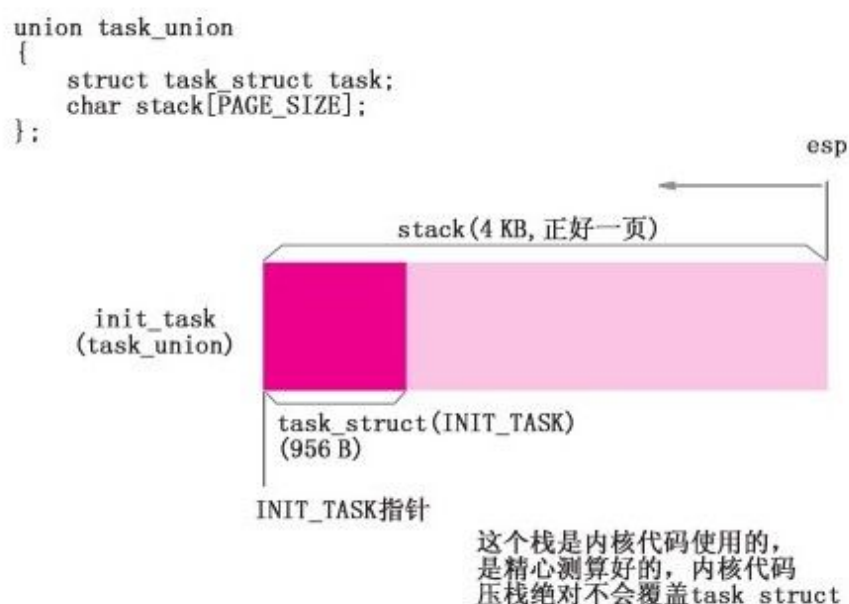


图 2-20 task_union结构示意图

`sched_init ()` 函数接下来用for循环将`task[64]`除进程0占用的0项外的其余63项清空，同时将GDT的TSS1、LDT1往上的所有表项清零，效果如图2-21所示。

初始化进程0相关的管理结构的最后一步是非常重要的，是将TR寄存器指向TSS0、LDTR寄存器指向LDT0，这样，CPU就能通过TR、

LDTR寄存器找到进程0的TSS0、LDT0，也能找到一切和进程0相关的管理信息。

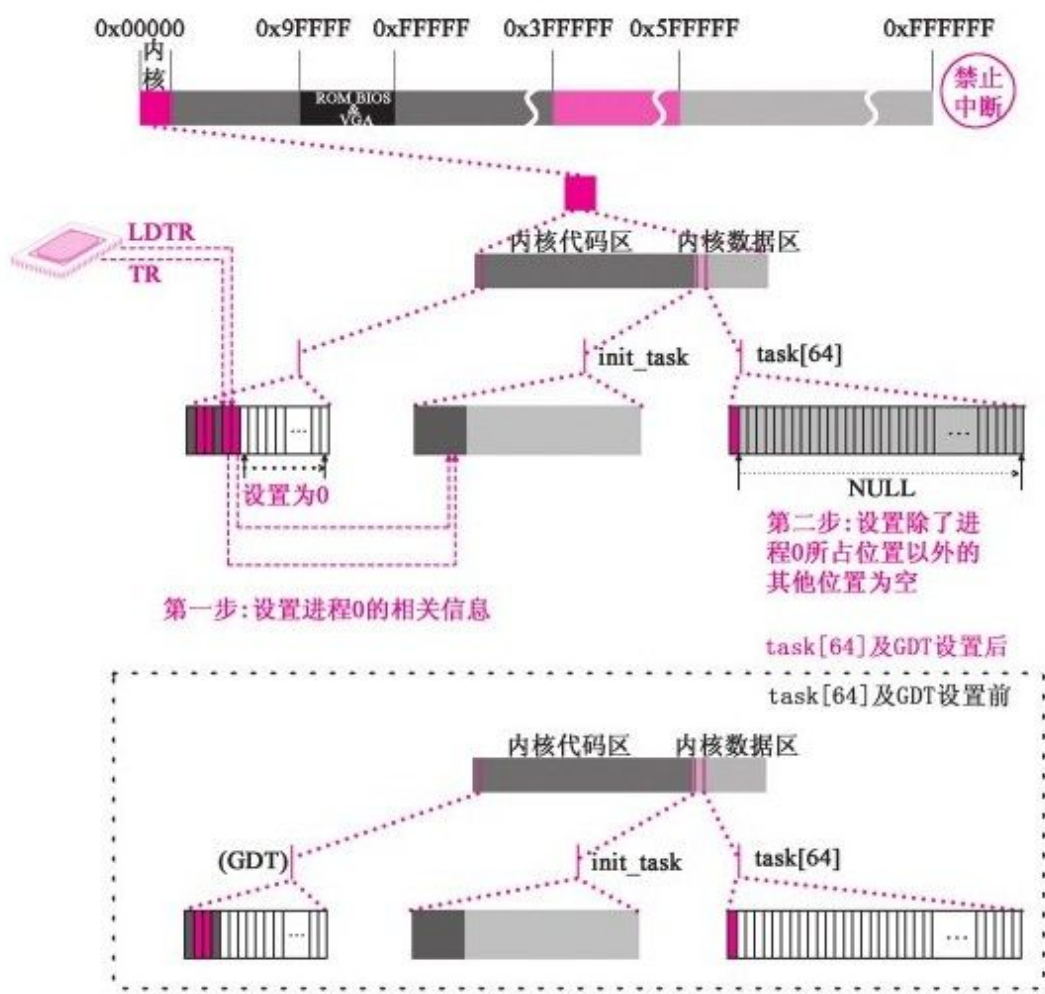


图 2-21 进程相关事务初始化设置

2.9.2 设置时钟中断

接下来就对时钟中断进行设置。时钟中断是进程0及其他由它创建的进程轮转的基础。对时钟中断进行设置的过程具体分为如下三个步骤。

1) 对支持轮询的8253定时器进行设置。这一步操作如图2-20中的第一步所示，其中LATCH最关键。LATCH是通过一个宏定义的，通过它在sched.c中的定义“`#define LATCH (1193180/HZ)`”，即系统每10毫秒发生一次时钟中断。

2) 设置时钟中断，如图2-22中的第二步所示，`timer_interrupt ()` 函数挂接后，在发生时钟

中断时，系统就可以通过IDT找到这个服务程序来进行具体的处理。

3) 将8259A芯片中与时钟中断相关的屏蔽码打开，时钟中断就可以产生了。从现在开始，时钟中断每1/100秒就产生一次。由于此时处于“关中断”状态，CPU并不响应，但进程0已经具备参与进程轮转的潜能。

2.9.3 设置系统调用总入口

将系统调用处理函数`system_call`与`int 0x80`中断描述符表挂接。`system_call`是整个操作系统中系统调用软中断的总入口。所有用户程序使用系统调用，产生`int 0x80`软中断后，操作系统都是通过这个总入口找到具体的系统调用函数。该过程如图2-23所示。

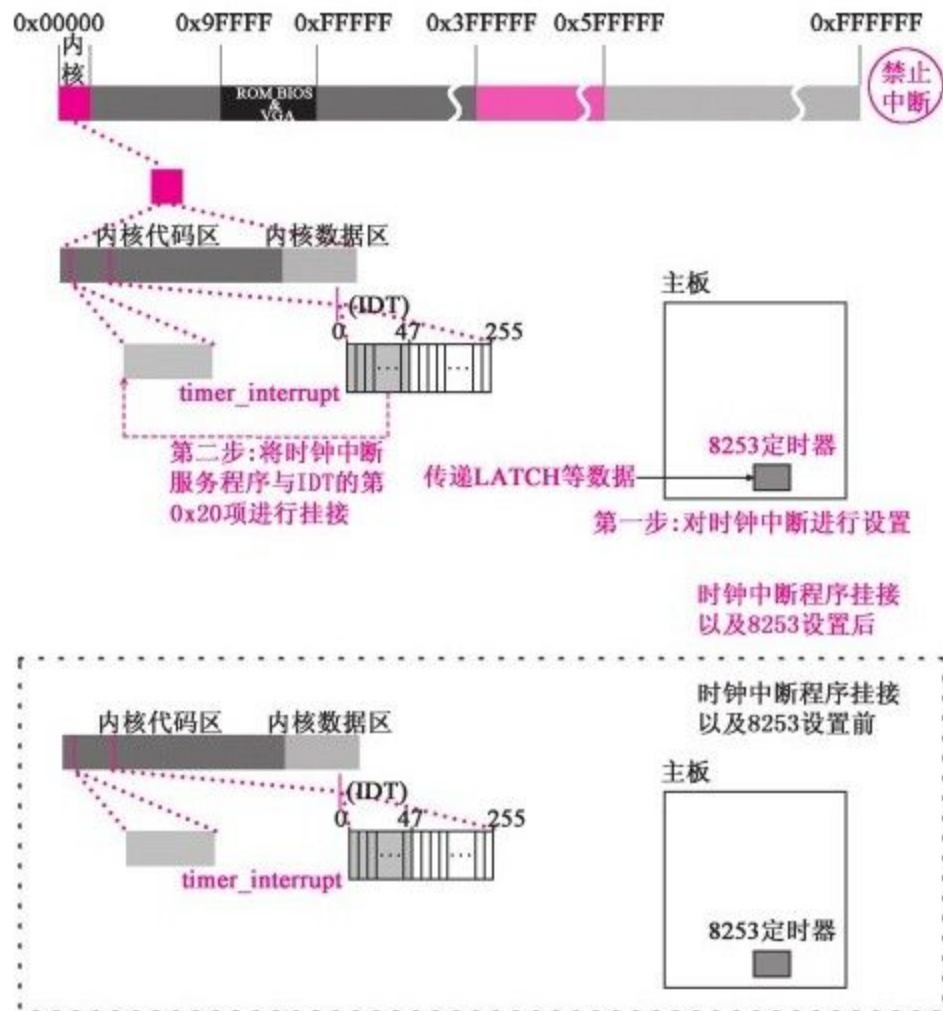


图 2-22 时钟中断设置

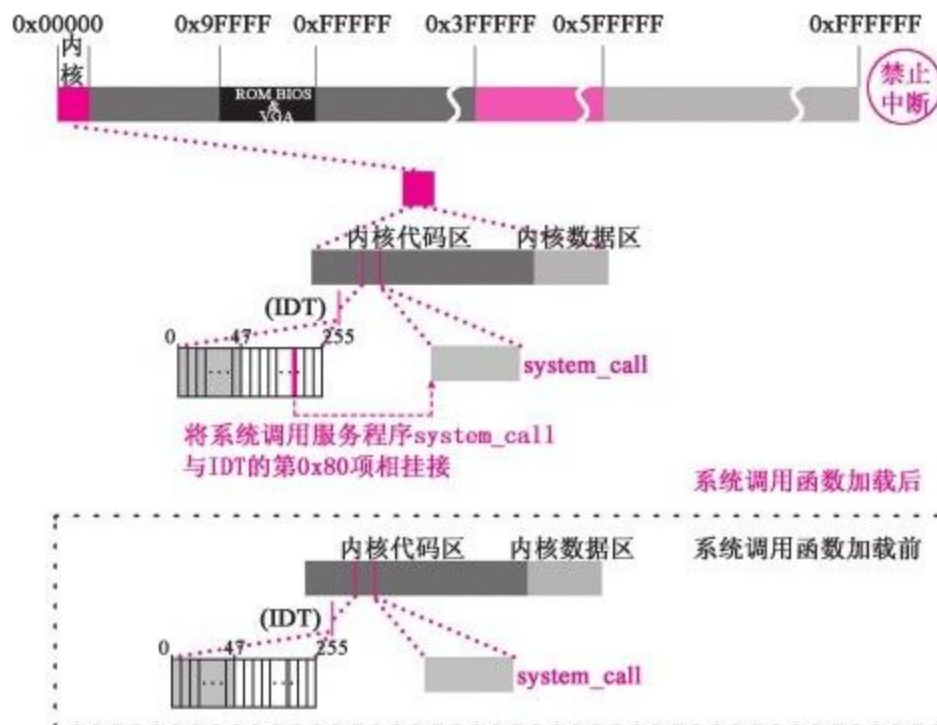


图 2-23 系统调用服务程序挂接

系统调用函数是操作系统对用户程序的基本支持。在操作系统中，依托硬件提供的特权级对内核进行保护，不允许用户进程直接访问内核代码。但进程有大量的像读盘、创建子进程之类的具体事务处理需要内核代码的支持。为了解决这个矛盾，操作系统的设计者提供了系统调用的解

决方案，提供一套系统服务接口。用户进程只要想和内核打交道，就调用这套接口程序，之后，就会立即引发int 0x80软中断，后面的事情就不需要用户程序管了，而是通过另一条执行路线——由CPU对这个中断信号响应，翻转特权级（从用户进程的3特权级翻转到内核的0特权级），通过IDT找到系统调用端口，调用具体的系统调用函数来处理事务，之后，再iret翻转回到进程的3特权级，进程继续执行原来的逻辑，这样矛盾就解决了。

2.10 初始化缓冲区管理结构

缓冲区是内存与外设（如硬盘，以后以硬盘为例）进行数据交互的媒介。内存与硬盘最大的区别在于，硬盘的作用仅仅是对数据信息以很低的成本做大量数据的断电保存，并不参与运算

（因为CPU无法到硬盘上进行寻址），而内存除了需要对数据进行保存以外，更重要的是要与CPU、总线配合进行数据运算。缓冲区则介于两者之间，它既对数据信息进行保存，也能够参与一些像查找、组织之类的间接、辅助性运算。有了缓冲区这个媒介以后，对外设而言，它仅需要考虑与缓冲区进行数据交互是否符合要求，而不需要考虑内存如何使用这些交互的数据；对内存而言，它也仅需要考虑与缓冲区交互的条件是否

成熟，而不需要关心此时外设对缓冲区的交互情况。两者的组织、管理和协调将由操作系统统一操作。

操作系统通过hash_table[NR_HASH]、buffer_head双向环链表组成的复杂的哈希表管理缓冲区。

操作系统通过调用buffer_init（）函数对缓冲区进行设置，执行代码如下：

```
//代码路径：init/main.c:

void main（void）

{

.....

buffer_init（buffer_memory_end）；

.....
```

```
}
```

在`buffer_init ()` 函数里，从内核的末端及缓冲区的末端同时开始，方向相对增长、配对地做出`buffer_head`、缓冲块，直到不足一对`buffer_head`、缓冲块。在第2章开始时设定的内存格局下，有3000多对`buffer_head`、缓冲块，`buffer_head`在低地址端，缓冲块在高地址端。

将`buffer_head`的成员设备号`b_dev`、引用次数`b_count`、“更新”标志`b_uptodate`、“脏”标志`b_dirt`、“锁定”标志`b_lock`设置为0。如图2-24所示，将`b_data`指针指向对应的缓冲块。利用`buffer_head`的`b_prev_free`、`b_next_free`，将所有的`buffer_head`形成双向链表。使`free_list`指向第一个

buffer_head，并利用free_list将buffer_head形成双向链表链接成双向环链表，如图2-25所示。

注意图2-26顶部所示的内存的变化。在紧靠系统内核的部分，多出了一块用黑色表示的内存区域，那里面存储的就是缓冲区管理结构。由于它管理着3000多个缓冲块，因此它占用的内存空间的大小，与内核几乎差不多。图2-26中也对空闲表的双向链表结构给出了形象的说明。

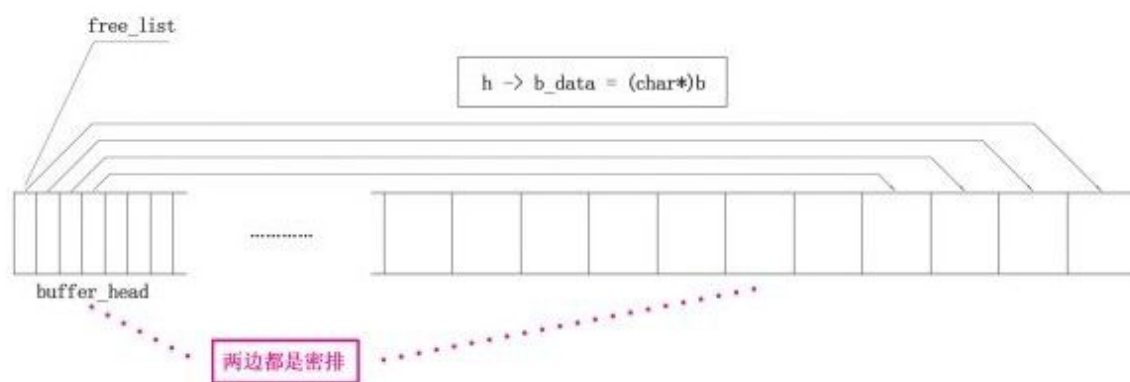


图 2-24 初始化示意图a

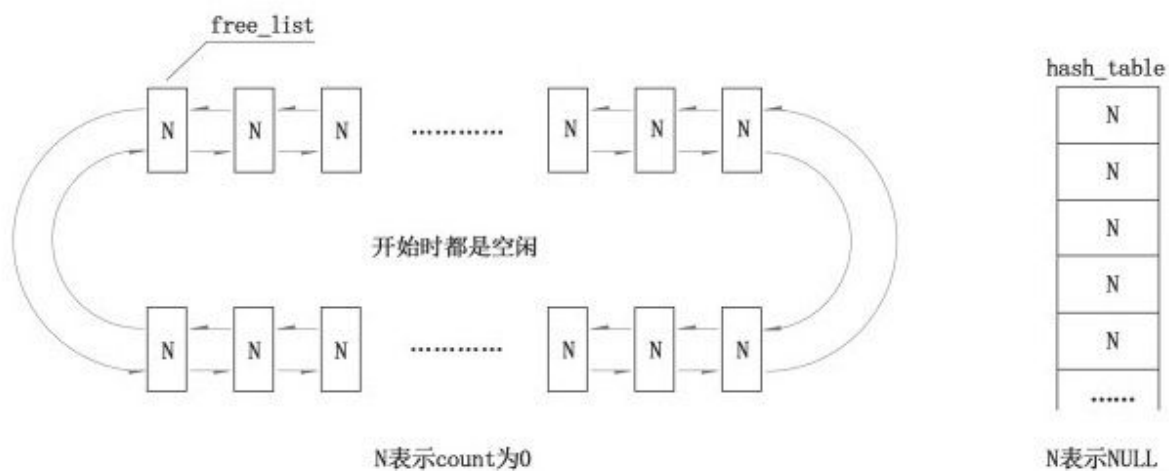


图 2-25 初始化示意图b

最后，对hash_table[307]进行设置，将hash_table[307]的所有项全部设置为NULL，如图2-26第二步所示。

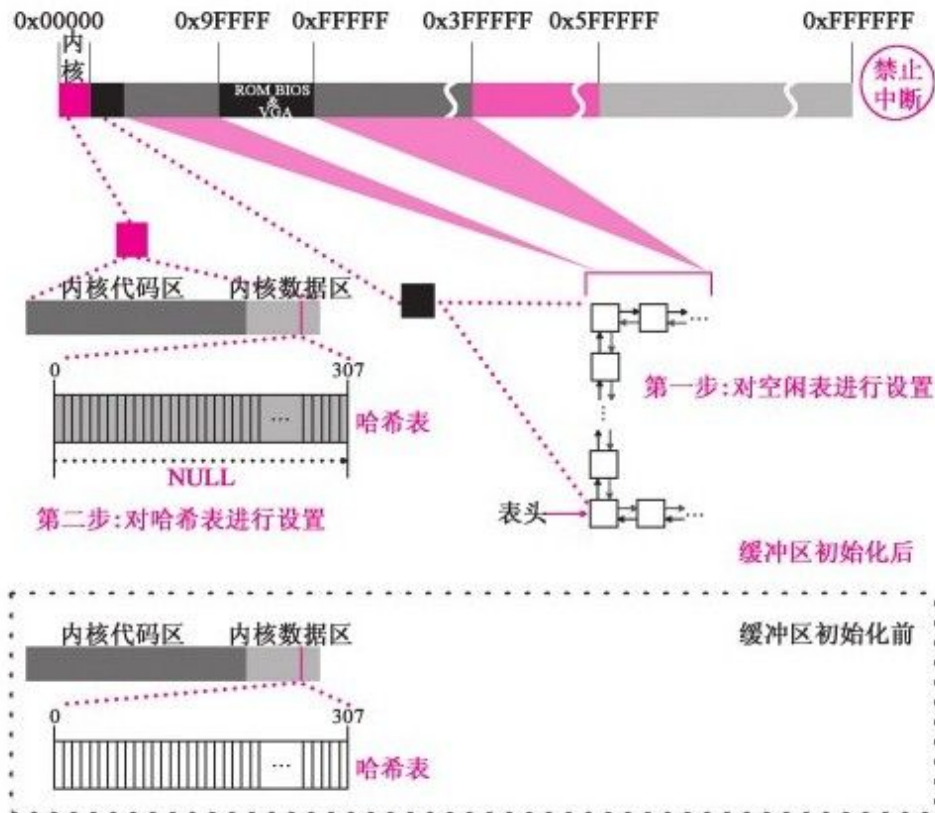


图 2-26 初始化缓冲区管理结构

对应的代码如下:

//代码路径: fs/buffer.c:

.....

```
struct buffer_head * start_buffer= (struct buffer_head *) &end;
```

```
struct buffer_head * hash_table[NR_HASH];
```

```
static struct buffer_head * free_list;
```

.....

```
void buffer_init (long buffer_end)
```

```
{
```

```
struct buffer_head * h=start_buffer;
```

```
void * b;
```

```
int i;
```

```
if (buffer_end==1 < < 20)
```

```
b= (void *) (640*1024) ;
```

```
else
```

```
b= (void *) buffer_end;
```

//h、b分别从缓冲区的低地址端和高地址端开始，每次对进
buffer_head、缓冲块各一个

```
//忽略剩余不足一对buffer_head、缓冲块的空间
```

```
while ( (b-=BLOCK_SIZE) >= ( (void *) (h+1) ) ) {
```

```
h->b_dev=0;
```

```
h->b_dirt=0;
```

```
h->b_count=0;
```

```
h->b_lock=0;
```

```

h->b_uptodate=0;

h->b_wait=NULL;

h->b_next=NULL; //这两项初始化为空，后续的使用将与
hash_table挂接

h->b_prev=NULL;

h->b_data= (char *) b; //每个buffer_head关联一个缓冲块

h->b_prev_free=h-1; //这两项使buffer_head分别与前、

h->b_next_free=h+1; //后buffer_head挂接，形成双向链表

h++;

NR_BUFFERS++;

if (b== (void *) 0x100000) //避开ROMBIOS&VGA

b= (void *) 0xA0000;

}

h--;

free_list=start_buffer; //free_list指向第一个buffer_head

free_list->b_prev_free=h; //使buffer_head双向链表

h->b_next_free=free_list; //形成双向环链表

for (i=0; i<NR_HASH; i++) //清空hash_table[307]

```

```
hash_table[i]=NULL;
```

```
}
```

注意看代码中`struct buffer_head *`

`h=start_buffer`这一行。这一行中的`start_buffer`确定了缓冲区的起始位置，这也就回答了2.2节中关于缓冲区起始点位置的这个问题。它是在`buffer.c`中定义的：

```
struct buffer_head * start_buffer= (s truct buffer_head *) &end;
```

这个`end`就是内核代码末端的地址。在代码编写阶段，设计者事先较难准确估算这个地址，于是就在内核模块链接期间设置`end`这个值，然后在这里使用。

2.11 初始化硬盘

硬盘的初始化为进程与硬盘这种块设备进行I/O通信建立了环境基础。

在hd_init（）函数中，将硬盘请求项服务程序do_hd_request（）与blk_dev控制结构相挂接，硬盘与请求项的交互工作将由do_hd_request（）函数来处理，然后将硬盘中断服务程序hd_interrupt（）与IDT相挂接，最后，复位主8259A int2的屏蔽位，允许从片发出中断请求信号，复位硬盘的中断请求屏蔽位（在从片上），允许硬盘控制器发送中断请求信号。

执行代码如下：

//代码路径: init/main.c:

```
void main (void)
```

```
{
```

```
.....
```

```
hd_init () ;
```

```
.....
```

```
}
```

释 //代码路径: kernel/blk_dev/hd.c: //与rd_init类似, 参看rd_init的注

```
void hd_init (void)
```

```
{
```

blk_dev[MAJOR_NR].request_fn=DEVICE_REQUEST; //挂接
do_hd_request ()

```
set_intr_gate (0x2E, &hd_interrupt) ; //设置硬盘中断
```

求 outb_p (inb_p (0x21) &0xfb, 0x21) ; //允许8259A发出中断请

```
outb (inb_p (0xA1) &0xbf, 0xA1) ; //允许硬盘发送中断请求
```

```
}
```

图2-27形象地给出了初始化过程。

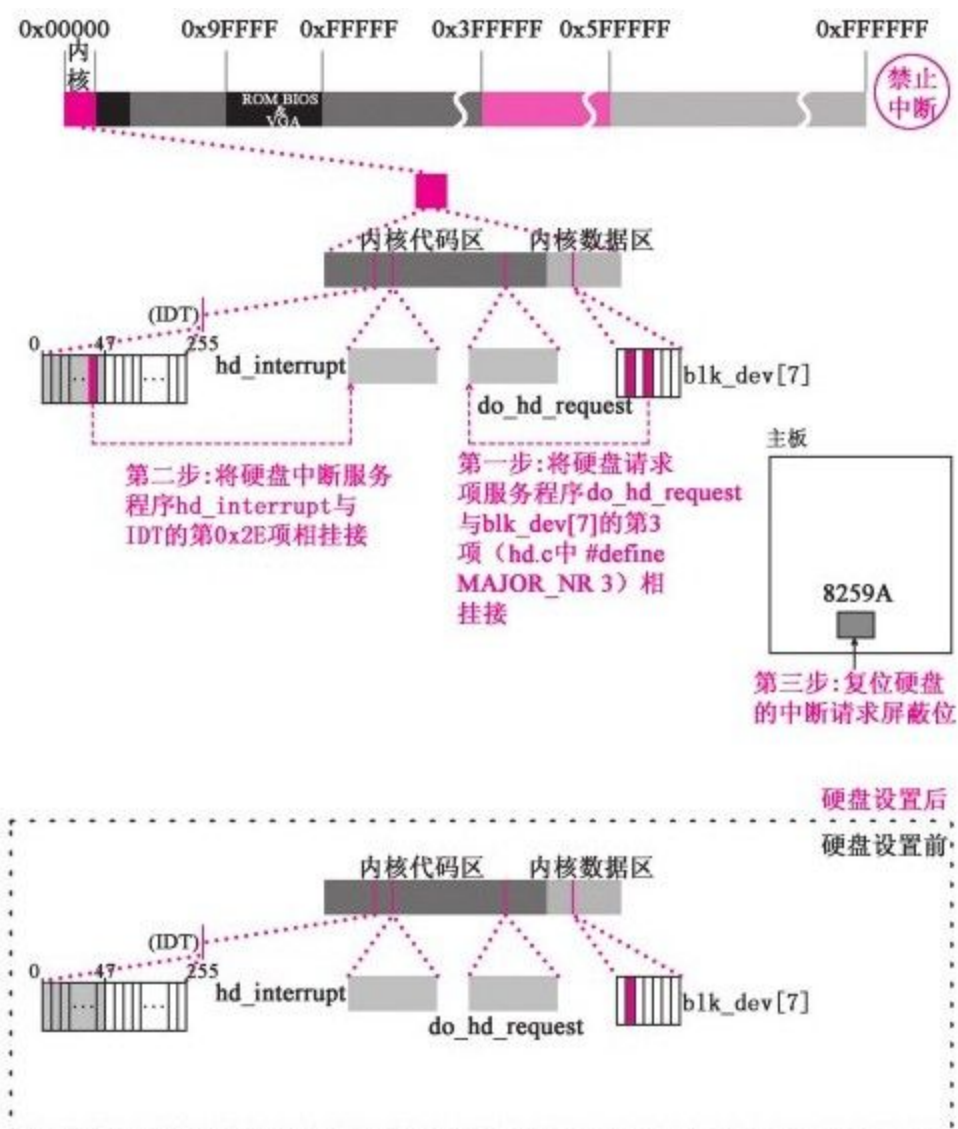


图 2-27 初始化硬盘

2.12 初始化软盘

软盘和软盘驱动器可以分离，合在一起才是一个整体。为了方便起见，本书所述的软盘除特别声明之外都是指软盘驱动器加软盘的整体。

软盘的初始化与硬盘的初始化类似，区别是挂接的函数是`do_fd_request`，初始化的是与软盘相关的中断。细节可参看初始化硬盘。

执行代码如下：

```
//代码路径：init/main.c:
```

```
void main (void)
```

```
{
```

```
.....
```



```
floppy_init () ; //与hd_init () 类似

.....

}

//代码路径: kernel/foppy.c:

.....

void floppy_init (void)

{

    blk_dev[MAJOR_NR].request_fn=DEVICE_REQUEST; //挂接
do_fd_request ()

    set_trap_gate (0x26, &floppy_interrupt) ; //设置软盘中断

    outb (inb_p (0x21) & ~0x40, 0x21) ; //允许软盘发送中断

}
```

图2-28给出了软盘初始化的主要步骤和完成后的效果。

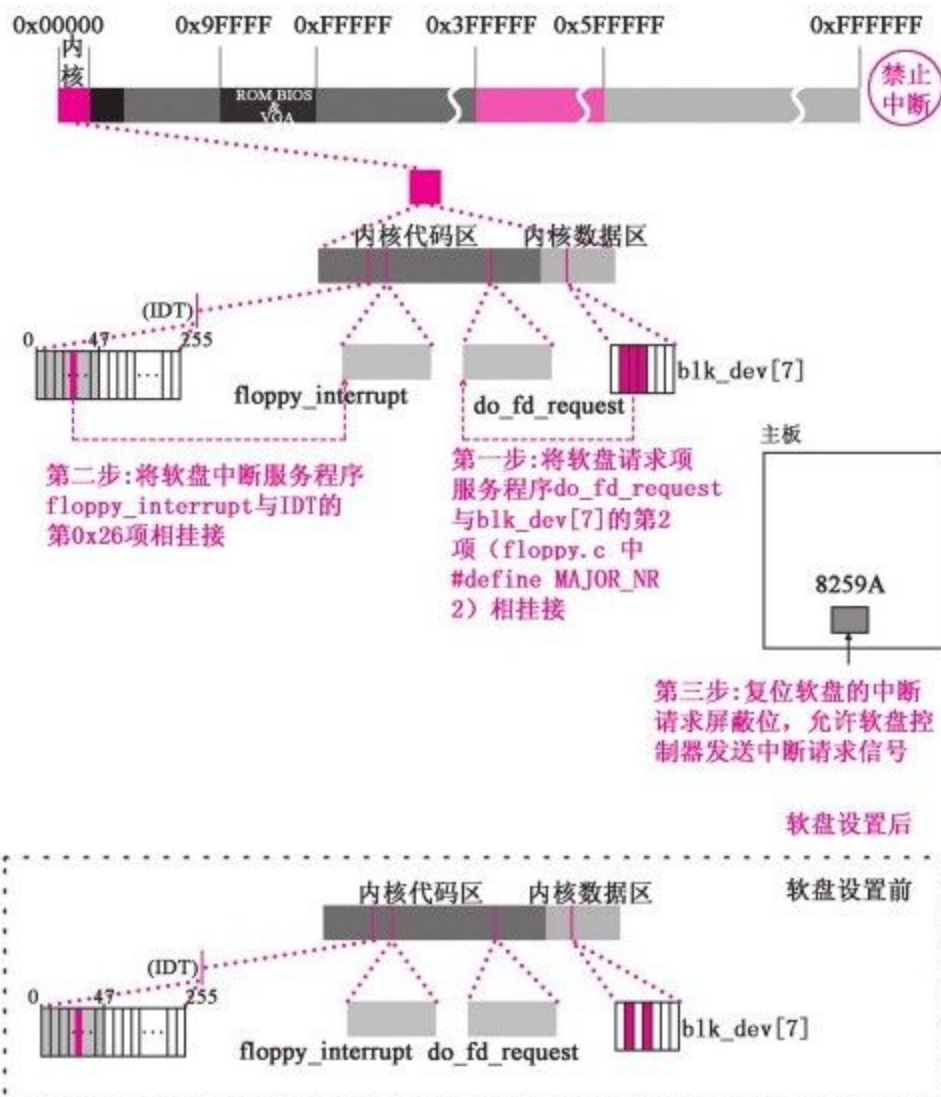


图 2-28 初始化软盘

2.13 开启中断

现在，系统中所有中断服务程序都已经和 IDT 正常挂接。这意味着中断服务体系已经构建完毕，系统可以在 32 位保护模式下处理中断，重要意义之一是可以使用系统调用。

可以开启中断了！

执行代码如下：

```
//代码路径：include/asm/system.h:
```

```
#define sti () __asm__ ("sti": )
```

```
//代码路径：init/main.c:
```

```
void main (void)
```

```
{
```

```

.....

sti ();

.....

}

```

图2-29给出了开中断后的效果，注意其中EFLAGS中的变化。

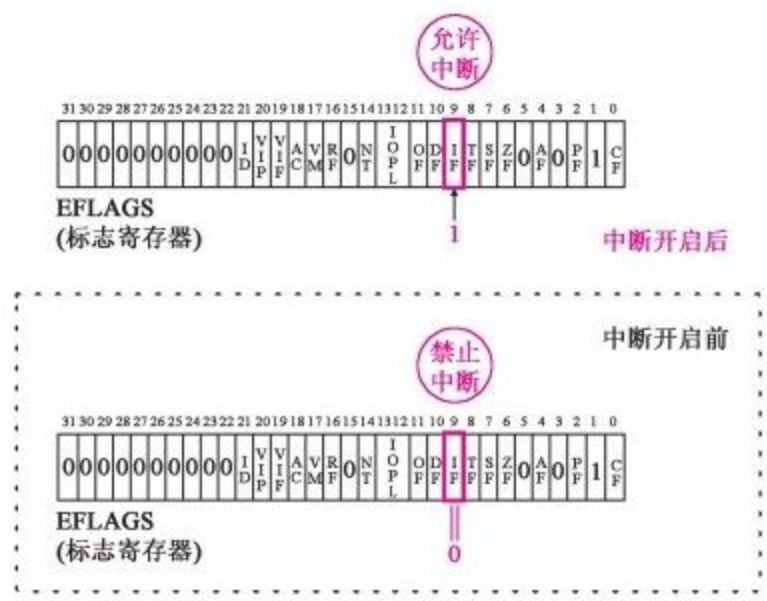


图 2-29 开启中断

2.14 进程0由0特权级翻转到3特权级，成为真正的进程

Linux操作系统规定，除进程0之外，所有进程都要由一个已有进程在3特权级下创建。在Linux 0.11中，进程0的代码和数据都是由操作系统的设计者写在内核代码、数据区，并且，此前处在0特权级，严格说还不是真正意义上的进程。为了遵守规则，在进程0正式创建进程1之前，要将进程0由0特权级转变为3特权级。方法是调用 `move_to_user_mode ()` 函数，模仿中断返回动作，实现进程0的特权级从0转变为3。

执行代码如下：

//代码路径: init/main.c:

```
void main (void)
```

```
{
```

```
.....
```

```
move_to_user_mode ();
```

```
.....
```

```
}
```

//代码路径: include/system.h: //参看1.3.4节

```
#define move_to_user_mode () \//模仿中断硬件压栈, 顺序是ss、  
esp、eflags、cs、eip
```

```
__asm__ ("movl%%esp, %%eax\n\t\"
```

```
"pushl$0x17\n\t"//SS进栈, 0x17即二进制的10111 (3特权级、  
LDT、数据段)
```

```
"pushl%%eax\n\t"//ESP进栈
```

```
"pushfl\n\t"//EFLAGS进栈
```

```
"pushl$0x0f\n\t"//CS进栈, 0x0f即1111 (3特权级、LDT、代码  
段)
```

```
"pushl$1f\n\t"//EIP进栈
```

```
"iret\n\t"//出栈恢复现场、翻转特权级从0到3
```

"1: \tmovl\$0x17, %%eax\n\t"V/下面的代码使ds、es、fs、gs与ss一致

```
"movw%%ax, %%ds\n\t"
```

```
"movw%%ax, %%es\n\t"
```

```
"movw%%ax, %%fs\n\t"
```

```
"movw%%ax, %%gs"
```

```
: »ax»)
```

IA-32体系结构翻转特权级的方法之一是用中断。第1章介绍过，当CPU接到中断请求时，能中断当前程序的执行序，将CS: EIP切换到相应的中断服务程序去执行，执行完毕又执行iret指令返回被中断的程序继续执行。

这期间，CPU硬件还做了两件事：一件是硬件保护现场和恢复现场，另一件是可以翻转特权级。

从代码的执行序上看，中断类似函数调用，都是从一段正在执行的代码跳转到另一段代码执行，执行之后返回原来的那段代码继续执行。为了保证执行完函数或中断服务程序的代码之后能准确返回原来的代码继续执行，需要在跳转到函数或中断服务程序代码之前，将起跳点的下一行代码的CS、EIP的值压栈保存，保护现场。函数或中断服务程序的代码执行完毕，再将栈中保存的值出栈给CS、EIP，此时的CS、EIP指向的就是起跳点的下一行，恢复现场。所以，CPU能准确地执行主调程序或被中断的程序。实际需要保护的寄存器还有EFLAGS等。

中断与函数调用不同的是，函数调用是程序员事先设计好的，知道在代码的哪个地方调用，

编译器可以预先编译出压栈保护现场和出栈恢复现场的代码；而中断的发生是不可预见的，无法预先编译出保护、恢复的代码，只好由硬件完成保护、恢复的压栈、出栈动作。所以，`int`指令会引发CPU硬件完成SS、ESP、EFLAGS、CS、EIP的值按序进栈，同理，CPU执行`iret`指令会将栈中的值自动按反序恢复给这5个寄存器。

CPU响应中断的时候，根据DPL的设置，可以实现指定的特权级之间的翻转。前面的`sched_init`函数中的`set_system_gate (0x80, &system_call)`就是设置的`int 0x80`中断由3特权级翻转到0特权级，3特权级的进程做了系统调用`int 0x80`，CPU就会翻转到0特权级执行系统代码。同

理，`iret`又会从0特权级的系统代码翻转回3特权级执行进程代码。

`move_to_user_mode`（）函数就是根据这个原理，利用`iret`实现从0特权级翻转到3特权级。

由于进程0的代码到现在一直处在0特权级，并不是从3特权级通过`int`翻转到0特权级的，栈中并没有`int`自动压栈的5个寄存器的值。为了`iret`的正确使用，设计者手工写压栈代码模拟`int`的压栈，当执行`iret`指令时，CPU自动将这5个寄存器的值按序恢复给CPU,CPU就会翻转到3特权级的段，执行3特权级的进程代码。

为了`iret`能翻转到3特权级，不仅手工模拟的压栈顺序必须正确，而且SS、CS的特权级还必须

正确。注意：栈中的SS值是0x17，用二进制表示就是00010111，最后两位表示3，是用户特权级，倒数第3位是1，表示从LDT中获取段描述符，第4～5位的10表示从LDT的第3项中得到进程栈段的描述符。

当执行iret时，硬件会按序将5个push压栈的数据分别出栈给SS、ESP、EFLAGS、CS、EIP。压栈顺序与通常中断返回时硬件的出栈动作一样，返回的效果也是一样的。

执行完move_to_user_mode（），相当于进行了一次中断返回，进程0的特权级从0翻转为3，成为名副其实的进程。

2.15 本章小结

本章开始执行以main（）函数为代表的用C语言编写的操作系统内核代码，内容涉及硬件初始化、为内核及进程的正确运行所做的初始化、激活进程0。

硬件初始化又可以分为两类：一类是与主机有关的硬件初始化，包括规划内存格局、设置及初始化缓冲区、设置及初始化虚拟盘、初始化mem_map、初始化缓冲区管理结构等；另一类是与外设有关的初始化，包括设置根设备、初始化软盘、初始化硬盘等。

为内核及进程的正确运行所做的初始化，包括中断服务程序的挂接、初始化进程0等。

最后就是用仿中断的方法将进程0的特权级由0翻转到3，实现激活进程0。

第3章 进程1的创建及执行

现在，计算机中已经有了一个名副其实的、3特权级的进程——进程0。下面我们要详细讲解进程0做的第一项工作——创建进程1。

3.1 进程1的创建

进程0现在处在3特权级状态，即进程状态。正式开始运行要做的第一件事就是作为父进程调用fork函数创建第一个子进程——进程1，这是父子进程创建机制的第一次实际运用。以后，所有进程都是基于父子进程创建机制由父进程创建出来的。

3.1.1 进程0创建进程1

在Linux操作系统中创建新进程的时候，都是由父进程调用fork函数来实现的。该过程如图3-1所示。

执行代码如下：

```
//代码路径: init/main.c:

.....

static inline_syscall0 (int,fork) //对应fork () 函数

static inline_syscall0 (int,pause)

static inline_syscall1 (int,setup,void*, BIOS)

.....

void main (void)

{

sti () ;

move _to_user_mode () ;

if (! fork () ) { /*we count on this going ok*/
```

```
init ( ) ;
```

```
}
```

```
/*
```

```
*NOTE ! For any other task'pause ( ) 'would mean we have to get a
```

```
*signal to awaken,but task0 is the sole exception (see'schedule ( ) ')
```

```
*as task 0 gets activated at every idle moment (when no other tasks
```

```
*can run) .For task0'pause ( ) 'just means we go check if some other
```

```
*task can run,and if not we return here.
```

```
*/
```

```
for ( ; ) pause ( ) ;
```

```
}
```

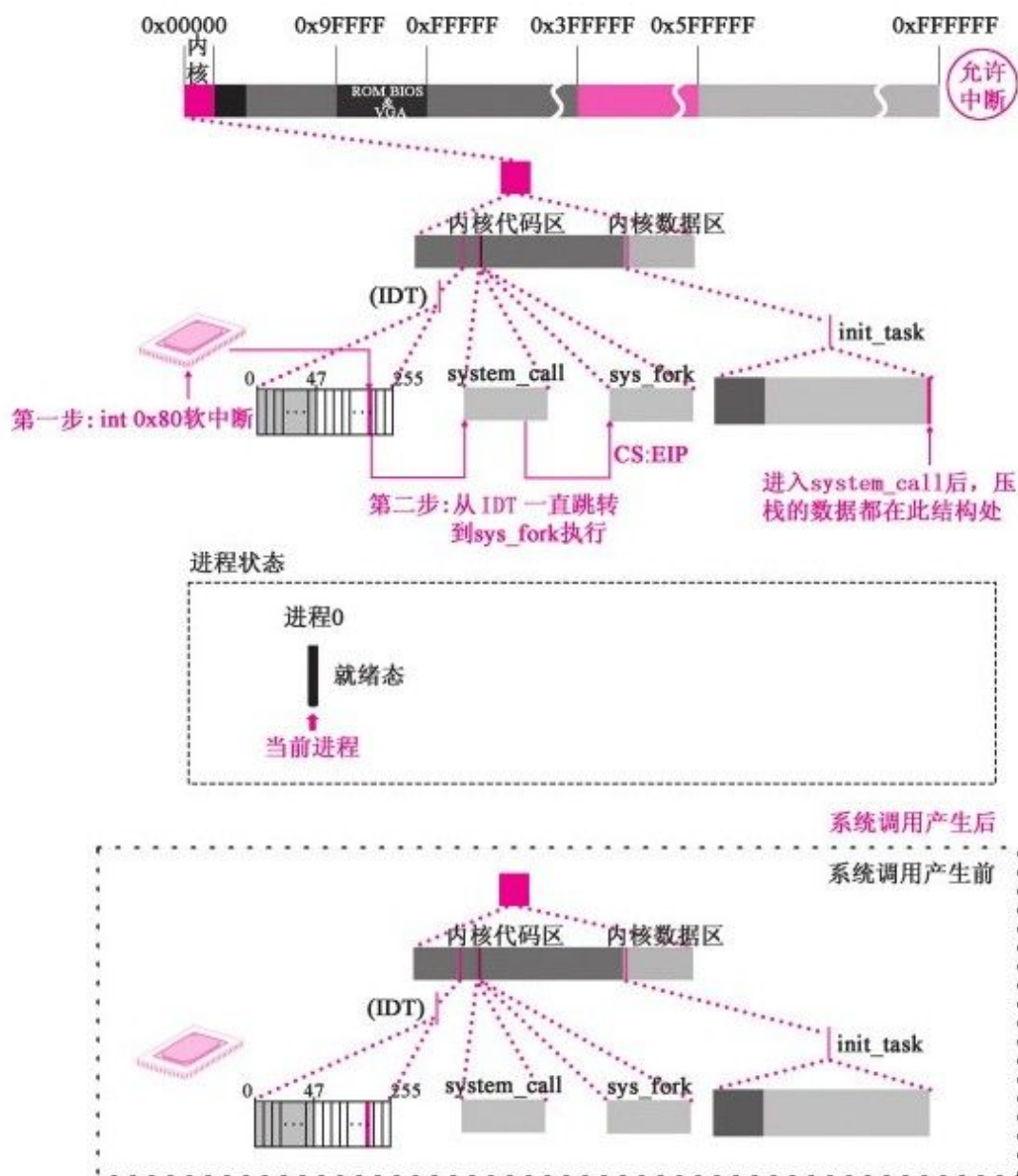


图 3-1 操作系统为创建进程1进行的准备工作

从上面main.c的代码中对fork () 的声明, 可知调用fork函数; 实际上是执行到unistd.h中的宏

函数syscall0中去，对应代码如下：

```
//代码路径： include/unistd.h:
```

```
.....
```

```
#define __NR_setup 0/*used only by init,to get system going*/
```

```
#define __NR_exit 1
```

```
#define __NR_fork 2
```

```
#define __NR_read 3
```

```
#define __NR_write 4
```

```
#define __NR_open 5
```

```
#define __NR_close 6
```

```
.....
```

```
#define syscall0 (type,name) \
```

```
type name (void) \
```

```
{\
```

```
long __res; \
```

```
__asm__volatile ("int$0x80"\
```

```

: "a" ( __res) \

: "0" ( __NR_##name) ) ; \

if ( __res >= 0) \

return (type) __res; \

errno=-__res; \

return-1; \

}

.....

volatile void _exit (int status) ;

int fcntl (int fildes,int cmd, ..... ) ;

int fork (void) ;

int getpid (void) ;

int getuid (void) ;

int geteuid (void) ;

.....

//代码路径: include/linux/sys.h:

extern int sys_setup ( ) ;

```

```
extern int sys_exit ( ) ;
```

extern int sys_fork () ; //对应system_call.s中的_sys_fork, 汇编中对应C语言的函数名在前面多加一个下划线"_", 如C语言的sys_fork对应汇编的就是_sys_fork

```
extern int sys_read ( ) ;
```

```
extern int sys_write ( ) ;
```

```
extern int sys_open ( ) ;
```

.....

```
fn _ptr sys_call_table[=  
{sys_setup,sys_exit,sys_fork,sys_read, //sys_fork对应_sys_call_table的第  
三项
```

```
sys_write,sys_open,sys_close,sys_waitpid,sys_creat,sys_link,
```

```
sys_unlink,sys_execve,sys_chdir,sys_time,sys_mknod,sys_chmod,
```

.....

syscall0展开后, 看上去像下面的样子:

释
int fork (void) //参看2.5节、2.9节、2.14节有关嵌入汇编的代码注

```
{
```

```
long __res;
```

`__asm__ volatile ("int$0x80"//int 0x80是所有系统调用函数的总入口, fork() 是其中之一, 参看2.9节的讲解及代码注释`

`: "=a" (__res) //第一个冒号后是输出部分, 将__res赋给eax`

`: "0" (__NR_fork)); //第二个冒号后是输入部分, "0": 同上寄存器, 即eax, __NR_fork就是2, 将2给eax`

`if (__res >= 0) //int 0x80中断返回后, 将执行这一句`

`return (int) __res;`

`errno=-__res;`

`return-1;`

`}`

`//重要: 别忘了int 0x80导致CPU硬件自动将ss、esp、eflags、cs、eip的值压栈! 参看2.14节的讲解及代码解释`

int 0x80的执行路线很长, 为了清楚起见, 将大致过程图示如下 (见图3-2)。

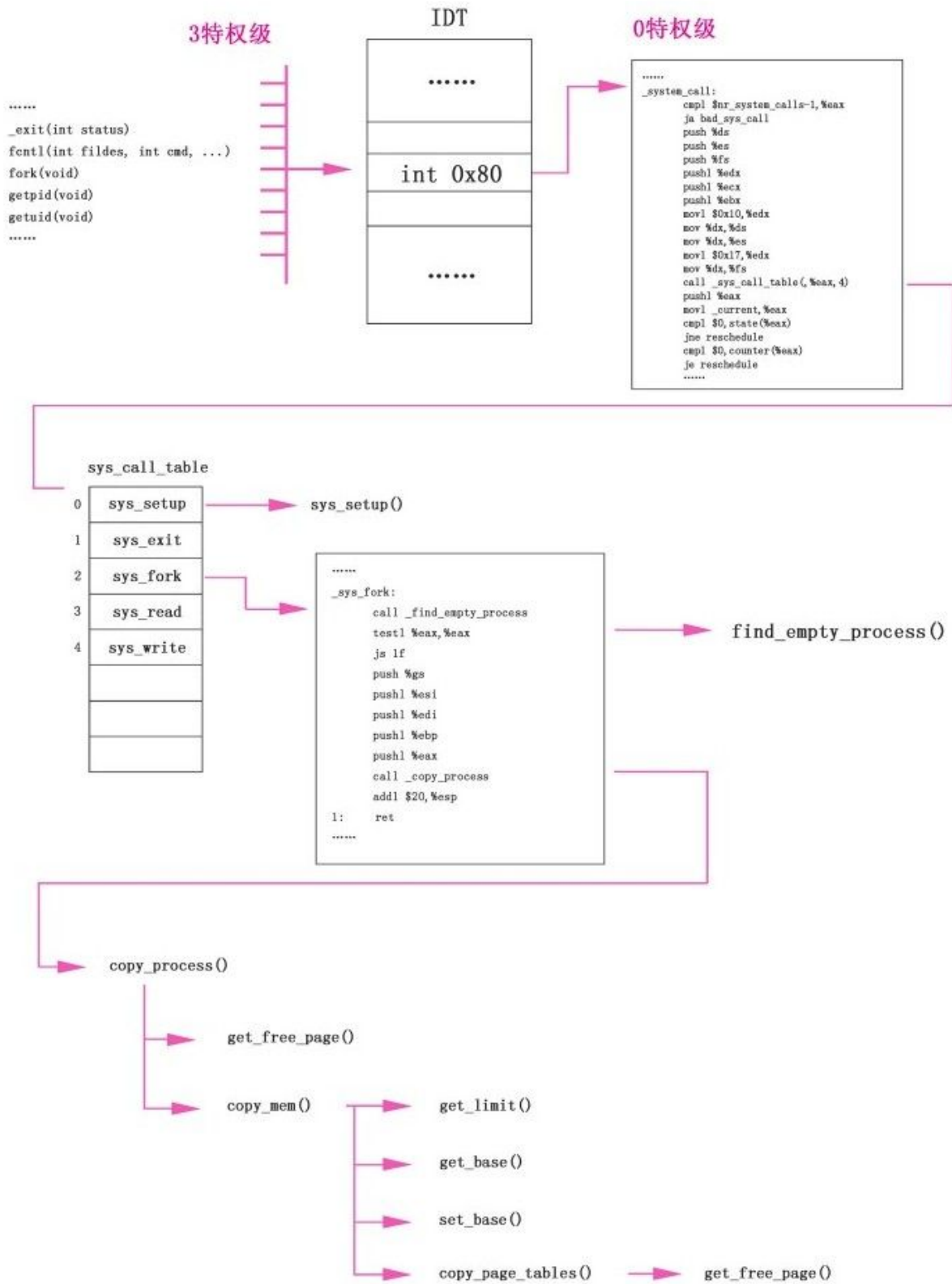


图 3-2 系统调用路线图

详细的执行步骤如下：

先执行："0"（__NR_fork）这一行，意思是将fork在sys_call_table[]中对应的函数编号__NR_fork（也就是2）赋值给eax。这个编号即sys_fork（）函数在sys_call_table中的偏移值。

紧接着就执行"int\$0x80"，产生一个软中断，CUP从3特权级的进程0代码跳到0特权级内核代码中执行。中断使CPU硬件自动将SS、ESP、EFLAGS、CS、EIP这5个寄存器的数值按照这个顺序压入图3-1所示的init_task中的进程0内核栈。注意其中init_task结构后面的红条，表示了刚刚压入内核栈的寄存器数值。前面刚刚提到的move_to_user_mode这个函数中做的压栈动作就是

模仿中断的硬件压栈，这些压栈的数据将在后续的`copy_process()`函数中用来初始化进程1的TSS。

值得注意，压栈的EIP指向当前指令"`int$0x80`"的下一行，即`if (__res >= 0)`这一行。这一行就是进程0从`fork`函数系统调用中断返回后第一条指令的位置。在后续的3.3节将看到，这一行也将是进程1开始执行的第一条指令位置。请记住这一点！

根据2.9节讲解的`sched_init`函数中`set_system_gate(0x80, &system_call)`的设置，CPU自动压栈完成后，跳转到`system_call.s`中的`_system_call`处执行，继续将DS、ES、FS、EDX、ECX、EBX压栈（以上一系列的压栈操作

都是为了后面调用`copy_process`函数中初始化进程1中的TSS做准备)。最终，内核通过刚刚设置的`eax`的偏移值“2”查询`sys_call_table[]`，得知本次系统调用对应的函数是`sys_fork()`。因为汇编中对应C语言的函数名在前面多加一个下划线“_”（如C语言的`sys_fork()`对应汇编的就是`_sys_fork`），所以跳转到`_sys_fork`处执行。

点评

一个函数的参数不是由函数定义的，而是由函数定义以外的程序通过压栈的方式“做”出来的，是操作系统底层代码与应用程序代码写作手法的差异之一；需要对C语言的编译、运行时结构非常清晰，才能彻底理解。运行时，C语言的参数存在于栈中。模仿这个原理，操作系统的设计者

可以将前面程序所压栈的值，按序“强行”认定为函数的参数；当call这个函数时，这些值就可以当做参数使用。

上述过程的执行代码如下：

```
//代码路径: kernel/system_call.s:
```

```
.....
```

```
_system_call: #int 0x80——系统调用的总入口
```

```
cmpl $nr_system_calls-1, %eax
```

```
ja bad_sys_call
```

push %ds#下面6个push都是为了copy_process () 的参数，请记住压栈的顺序，别忘了前面的int 0x80还压了5个寄存器的值进栈

```
push %es
```

```
push %fs
```

```
pushl %edx
```

```
pushl %ecx#push%ebx, %ecx, %edx as parameters
```

```
pushl %ebx#to the system call
```

movl \$0x10, %edx#set up ds,es to kernel space

mov %dx, %ds

mov %dx, %es

movl \$0x17, %edx#fs points to local data space

mov %dx, %fs

call _sys_call_table (, %eax, 4) #eax是2, 可以看成call
(_sys_call_table+2×4) 就是_sys_fork的入口

pushl %eax

movl _current, %eax

cmpl \$0, state (%eax) #state

jne reschedule

cmpl \$0, counter (%eax) #counter

je reschedule

ret _from_sys_call:

movl _current, %eax#task[0]cannot have signals

cmpl _task, %eax

je 3f

cmpw \$0x0f,CS (%esp) #was old code segment supervisor?

jne 3f

cmpw \$0x17, OLDSS (%esp) #was stack segment=0x17?

jne 3f

movl signal (%eax) , %ebx

movl blocked (%eax) , %ecx

notl %ecx

andl %ebx, %ecx

bsfl %ecx, %ecx

je 3f

btrl %ecx, %ebx

movl %ebx,signal (%eax)

incl %ecx

pushl %ecx

call _do_signal

popl %eax

3: popl%eax

popl %ebx

```
popl %ecx  
  
popl %edx  
  
pop %fs  
  
pop %es  
  
pop %ds  
  
iret  
  
.....  
  
_sys_fork: #sys_fork函数的入口  
  
.....
```

call_sys_call_table (, %eax, 4) 中的eax是2, 这一行可以看成call_sys_call_table+2×4 (4的意思是_sys_call_table[]的每一项有4字节), 相当于call_sys_call_table[2] (见图3-1的左中部分), 就是执行sys_fork。

注意： `call_sys_call_table (, %eax, 4)` 指令本身也会压栈保护现场，这个压栈体现在后面 `copy_process` 函数的第6个参数 `long none` 。

对应代码如下：

```
//代码路径： kernel/system_call.s:
```

```
.....
```

```
_system_call:
```

```
.....
```

```
_sys_fork:
```

```
call _fnd_empty_process#调用find_empty_process ()
```

```
testl %eax, %eax#如果返回的是-EAGAIN (11)，说明已有64个  
进程在运行
```

```
js 1f
```

```
push %gs#5个push也作为copy_process () 的参数初始
```

```
pushl %esi
```

```
pushl %edi
```

```
pushl %ebp
```

```
pushl %eax
```

```
call _copy_process#调用copy_process ()
```

```
addl $20, %esp
```

```
1: ret
```

```
.....
```

3.1.2 在task[64]中为进程1申请一个空闲位置并获取进程号

开始执行sys_fork（）。

前面2.9节介绍过，在sched_init（）函数中已经对task[64]除0项以外的所有项清空。现在调用find_empty_process（）函数为进程1获得一个可用的进程号和task[64]中的一个位置。图3-3标示了这个调用的效果。

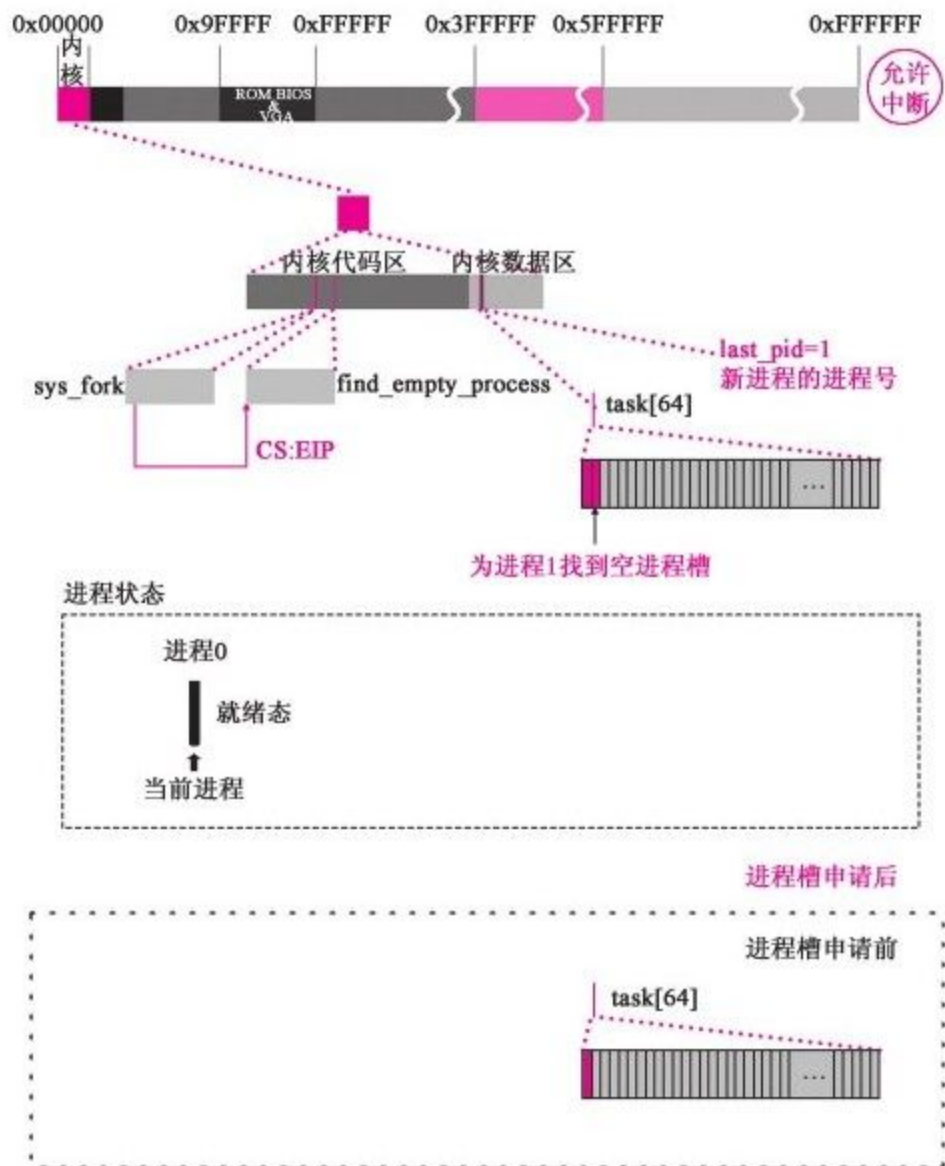


图 3-3 在内核数据区中查找进程空闲项

在 `find_empty_process()` 函数中，内核用全局变量 `last_pid` 来存放系统自开机以来累计的进程

数，也将此变量用作新建进程的进程号。内核第一次遍历task[64]，“&&”条件成立说明last_pid已被使用，则++last_pid，直到获得用于新进程的进程号。第二次遍历task[64]，获得第一个空闲的i，俗称任务号。

现在，两次遍历的结果是新的进程号last_pid就是1，在task[64]中占据第二项。图3-3标示了这个结果。

因为Linux 0.11的task[64]只有64项，最多只能同时运行64个进程，如果find_empty_process

() 函数返回-EAGAIN，意味着当前已经有64个进程在运行，当然这种情况现在还不会发生。执行代码如下：

//代码路径: kernel/fork.c:

.....

long last_pid=0;

.....

int find_empty_process (void) //为新创建的进程找到一个空闲的位置, NR_TASKS是64

{

int i;

repeat:

1 if ((++last_pid) < 0) last_pid=1; //如果++后last_pid溢出, 则置

for (i=0; i<NR_TASKS; i++) //现在, ++后last_pid为1。找到有效的last_pid

if (task[i]&&task[i]->pid==last_pid) goto repeat;

for (i=1; i<NR_TASKS; i++) //返回第一个空闲的i

if (! task[i])

return i;

return-EAGAIN; //EAGAIN是11

}

进程1的进程号及在task[64]中的位置确定后，正在创建的进程1就等于有了身份。接下来，在进程0的内核栈中继续压栈，将5个寄存器值进栈，为调用copy_process（）函数准备参数，这些数据也是用来初始化进程1的TSS。注意：最后压栈的eax的值就是find_empty_process（）函数返回的任务号，也将是copy_process（）函数的第一个参数int nr。

压栈结束后，开始调用copy_process（）函数，如图3-4中第二步所示。

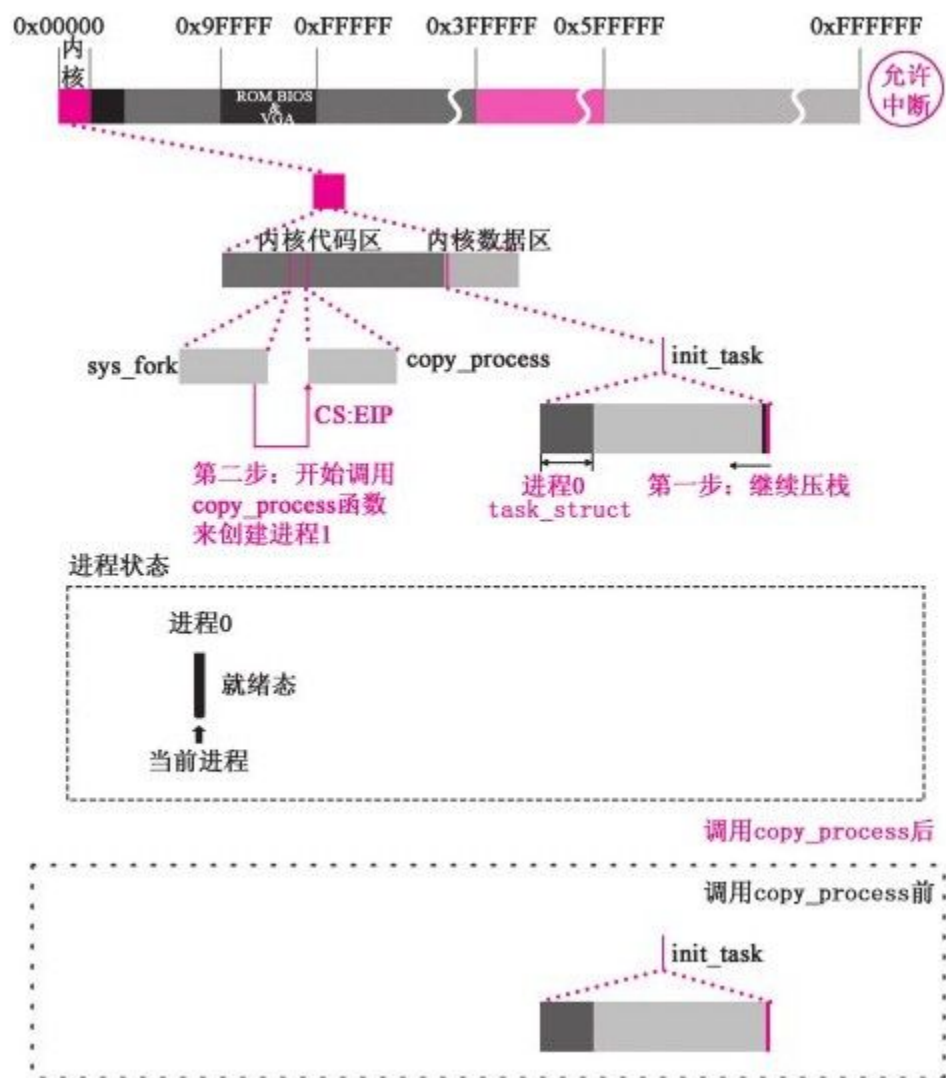


图 3-4 调用 `copy_process` () 之前的压栈动作

3.1.3 调用copy_process函数

进程0已经成为一个可以创建子进程的父进程，在内核中有“进程0的task_struct”和“进程0的页表项”等专属进程0的管理信息。进程0将在copy_process（）函数中做非常重要的、体现父子进程创建机制的工作：

- 1) 为进程1创建task_struct，将进程0的task_struct的内容复制给进程1。
- 2) 为进程1的task_struct、tss做个性化设置。
- 3) 为进程1创建第一个页表，将进程0的页表项内容赋给这个页表。
- 4) 进程1共享进程0的文件。

5) 设置进程1的GDT项。

6) 最后将进程1设置为就绪态，使其可以参与进程间的轮转。

现在调用copy_process () 函数！

在讲解copy_process () 函数之前，值得提醒的是，所有的参数都是前面的代码累积压栈形成的，这些参数的数值都与压栈时的状态有关。执行代码如下：

//代码路径： kernel/fork.c:

```
int copy_process (int nr,long ebp,long edi,long esi,long gs,long  
none,  
  
long ebx,long ecx,long edx,  
  
long fs,long es,long ds,  
  
long eip,long cs,long efags,long esp,long ss)
```

//注意：这些参数是int 0x80、system_call、sys_fork多次累积压栈的结果，顺序是完全一致的

```
{  
  
    struct task_struct * p;  
  
    int i;  
  
    struct file * f;
```

//在16 MB内存的最高端获取一页，强制类型转换的潜台词是将这个页当task_union用，参看2.9节

```
    p= (struct task_struct *) get_free_page ();  
  
    if (! p)  
  
        return-EAGAIN;
```

task[nr]=p; //此时的nr就是1，潜台词是将这个页当task_union用，参看2.9节

```
    .....  
  
}
```

进入copy_process () 函数后，调用
get_free_page () 函数，在主内存申请一个空闲

页面，并将申请到的页面清零，用于进程1的 task_struct及内核栈。

按照get_free_page（）函数的算法，是从主内存的末端开始向低地址端递进，现在是开机以来，操作系统内核第一次为进程在主内存申请空闲页面，申请到的空闲页面肯定在16 MB主内存的最末端。

执行代码如下：

//代码路径：mm/memory.c:

unsigned long get_free_page（void）//遍历mem map[]，找到主内存中（从高地址开始）第一个空闲页面

{//参看前面的嵌入汇编的代码注释

register unsigned long__res asm（"ax"）；

__asm__（"std; repne; scasb\n\t"//反向扫描串（mem map[]），al（0）与di不等则重复（找引用对数为0的项）

"jne 1f\n\t"//找不到空闲页，跳转到1

"movb\$1, 1 (%%edi) \n\t"//将1赋给edi+1的位置，在mem map[]中，将找到0的项的引用计数置为1

"sall\$12, %%ecx\n\t"//ecx算数左移12位，页的相对地址

"addl%2, %%ecx\n\t"//LOW MEN+ecx，页的物理地址

"movl%%ecx, %%edx\n\t"

"movl\$1024, %%ecx\n\t"

"leal 4092 (%%edx) , %%edi\n\t"//将edx+4 KB的有效地址赋给edi

"rep; stosl\n\t"//将eax（即"0"（0））赋给edi指向的地址，目的是页面清零

"movl%%edx, %%eax\n"

"1: "

: "=a" (__res)

: "0" (0) , "i" (LOW_MEM) , "c" (PAGING_PAGES) ,

"D" (mem_map+PAGING_PAGES-1) //edx,mem map[]的最后一个元素

: "di", "cx", "dx") ; //第三个冒号后是程序中改变过的量

return __res;

}

回到`copy_process`函数，将这个页面的指针强制类型转换为指向`task_struct`的指针类型，并挂载在`task[1]`上，即`task[nr]=p`。nr就是第一个参数，是`find_empty_process`函数返回的任务号。

请注意，C语言中的指针有地址的含义，更有类型的含义！强制类型转换的意思是“认定”这个页面的低地址端就是进程1的`task_struct`的首地址，同时暗示了高地址部分是内核栈。了解了这一点，后面的`p->tss.esp0=PAGE_SIZE+ (long) p`就不奇怪了。

点评

`task_struct`是操作系统标识、管理进程的最重要的数据结构，每一个进程必须具备只属于自己的、唯一的`task_struct`。

```
//代码路径: kernel/fork.c:
```

```
int copy_process (int nr,long ebp,long edi,long esi,long gs,long  
none,
```

```
long ebx,long ecx,long edx,
```

```
long fs,long es,long ds,
```

```
long eip,long cs,long efags,long esp,long ss) {
```

```
.....
```

```
if (! p)
```

```
return-EAGAIN;
```

```
task[nr]=p; //此时的nr就是1
```

```
/*current指向当前进程的task_struct的指针，当前进程是进程0。下面这行的意思：将父进程的task_struct赋给子进程。这是父子进程创建机制的重要体现。这行代码执行后，父子进程的task_struct将完全一样*/
```

```
*p=*current; /*NOTE ! this doesn't copy the supervisor stack*/
```

/*重要！注意指针类型，只复制task_struct，并未将4 KB都复制，即进程0的内核栈并未复制*/

p->state=TASK_UNINTERRUPTIBLE; //只有内核代码中明确表示将该进程设置为就绪状态才能被唤醒，除此之外，没有任何办法将其唤醒

p->pid=last_pid; //开始子进程的个性化设置

p->father=current->pid;

p->counter=p->priority;

p->signal=0;

p->alarm=0;

p->leader=0; /*process leadership doesn't inherit*/

p->utime=p->stime=0;

p->cutime=p->cstime=0;

p->start_time=jiffies;

p->tss.back_link=0; //开始设置子进程的TSS

.....

}

效果如图3-5（为了方便阅读，我们把2.9节的图2-20复制在下面）所示。

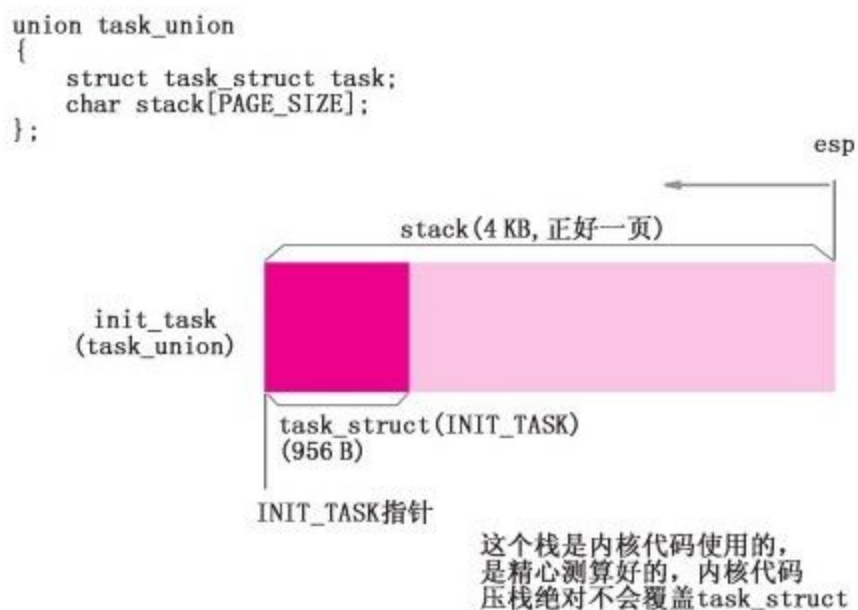


图 3-5 task_union结构示意图

点评

task_union的设计颇具匠心。前面是task_struct，后面是内核栈，增长的方向正好相反，正好占用一页，顺应分页机制，分配内存非

常方便。而且操作系统设计者肯定经过反复测试，保证内核代码所有可能的调用导致压栈的最大长度都不会覆盖前面的`task_struct`。因为内核代码都是操作系统设计者设计的，可以做到心中有数。相反，假如这个方法为用户进程提供栈空间，恐怕要出大问题了。

接下来的代码意义重大：

```
*p=*current; /*NOTE ! this doesn't copy the supervisor stack*/
```

`current`是指向当前进程的指针；`p`是进程1的指针。当前进程是进程0，是进程1的父进程。将父进程的`task_struct`复制给子进程，就是将父进程最重要的进程属性复制给了子进程，子进程继承

了父进程的绝大部分能力。这是父子进程创建机制的特点之一。

进程1的task_struct的雏形此时已经形成了，进程0的task_struct中的信息并不一定全都适用于进程1，因此还需要针对具体情况进行调整。初步设置进程1的task_struct如图3-6所示。从p->开始的代码，都是为进程1所做的个性化调整设置，其中调整TSS所用到的数据都是前面程序累积压栈形成的参数。

执行代码如下：

```
//代码路径: kernel/fork.c:

int copy_process (int nr,long ebp,long edi,long esi,long gs,long
none,

long ebx,long ecx,long edx,
```



```
long fs,long es,long ds,
```

```
long eip,long cs,long efags,long esp,long ss)
```

```
{
```

```
.....
```

```
p->start_time=jiffies;
```

```
p->tss.back_link=0; //开始设置子进程的TSS
```

```
p->tss.esp0=PAGE_SIZE+ (long) p; //esp0是内核栈指针，参看  
上面的注释及2.9.1节
```

```
p->tss.ss0=0x10; //0x10就是10000，0特权级，GDT，数据段
```

```
p->tss.eip=eip; //重要！就是参数的EIP，是int 0x80压栈的，指向  
的是：if (__res>=0)
```

```
p->tss.eflags=eflags;
```

```
p->tss.eax=0; //重要！决定main () 函数中if (! fork () ) 后面  
的分支走向
```

```
p->tss.ecx=ecx;
```

```
p->tss.edx=edx;
```

```
p->tss.ebx=ebx;
```

```
p->tss.esp=esp;
```

```
p->tss.ebp=ebp;
```

```
p->tss.esi=esi;

p->tss.edi=edi;

p->tss.es=es&0xffff;

p->tss.cs=cs&0xffff;

p->tss.ss=ss&0xffff;

p->tss.ds=ds&0xffff;

p->tss.fs=fs&0xffff;

p->tss.gs=gs&0xffff;

p->tss.ldt=_LDT (nr) ; //挂接子进程的LDT

p->tss.trace_bitmap=0x80000000;

if (last_task_used_math==current)

__asm__ („clts; fnsave%0": "m" (p->tss.i387) ) ;

.....

}
```

点评

```
p->tss.eip=eip;
```

```
p->tss.eax=0;
```

这两行代码为第二次执行fork（）中的if（__res>=0）埋下伏笔。这个伏笔比较隐讳，不太容易看出来，请读者一定要记住这件事！

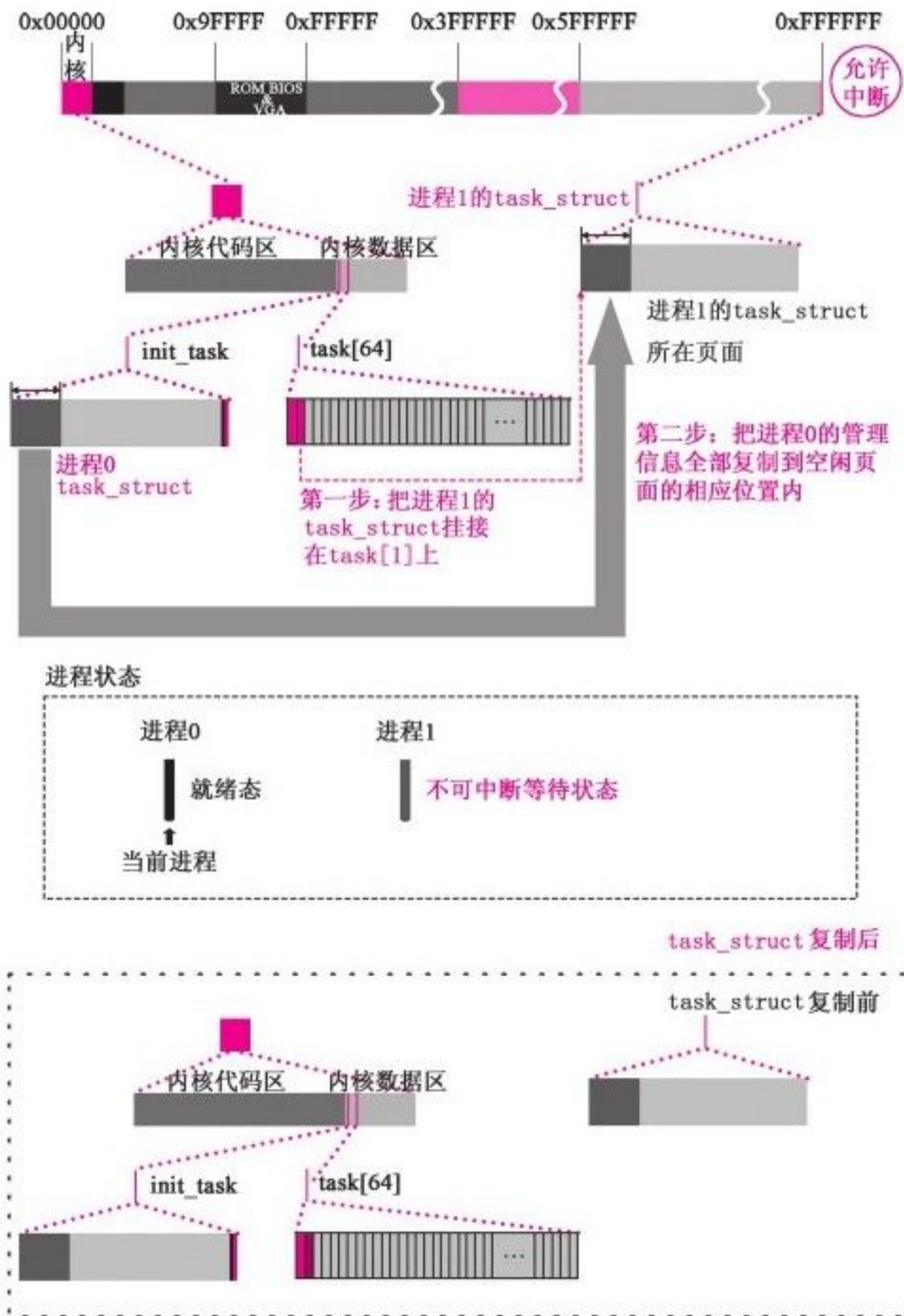


图 3-6 初步设置进程1的task_struct

调整完成后，进程1的task_struct如图3-7所示。

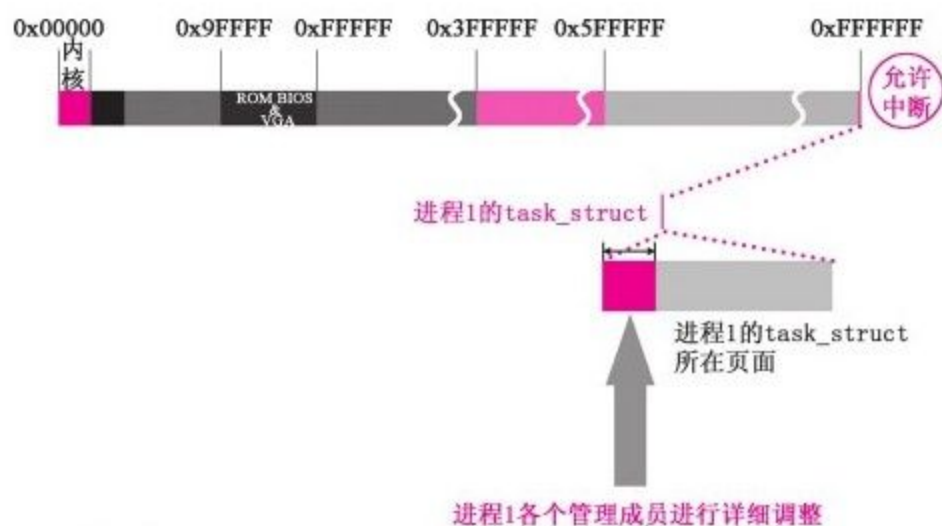


图 3-7 对进程1的task_struct进行调整



图 3-7 (续)

3.1.4 设置进程1的分页管理

Intel 80x86体系结构分页机制是基于保护模式的，先打开pe，才能打开pg，不存在没有pe的pg。保护模式是基于段的，换句话说，设置进程1的分页管理，就要先设置进程1的分段。

一般来讲，每个进程都要加载属于自己的代码、数据。这些代码、数据的寻址都是用段加偏移的形式，也就是逻辑地址形式表示的。CPU硬件自动将逻辑地址计算为CPU可寻址的线性地址，再根据操作系统对页目录表、页表的设置，自动将线性地址转换为分页的物理地址。操作系统正是沿着这个技术路线，先在进程1的64 MB线

性地址空间中设置代码段、数据段，然后设置页表、页目录。

1.在进程1的线性地址空间中设置代码段、数据段

调用`copy_mem()`函数，先设置进程1的代码段、数据段的段基址、段限长，提取当前进程（进程0）的代码段、数据段以及段限长的信息，并设置进程1的代码段和数据段的基地址。这个基地址就是它的进程号`nr*64 MB`。设置新进程LDT中段描述符中的基地址，如图3-8中的第一步所示。

执行代码如下：

```
//代码路径: kernel/fork.c:
```



```

int copy_process (int nr,long ebp,long edi,long esi,long gs,long
none,

long ebx,long ecx,long edx,

long fs,long es,long ds,

long eip,long cs,long efags,long esp,long ss)

{

.....

if (last_task_used_math==current)

__asm__ ("clts; fnsave%0": "m" (p->tss.i387) ) ;

if (copy_mem (nr,p) ) { //设置子进程的代码段、数据段及创
建、复制子进程的第一个页表

task[nr]=NULL; //现在不会出现这种情况

free_page ( (long) p) ;

return-EAGAIN;

}

for (i=0; i<NR_OPEN; i++) //下面将父进程相关文件属性的引
用计数加1，表明父子进程共享文件

.....

}

```

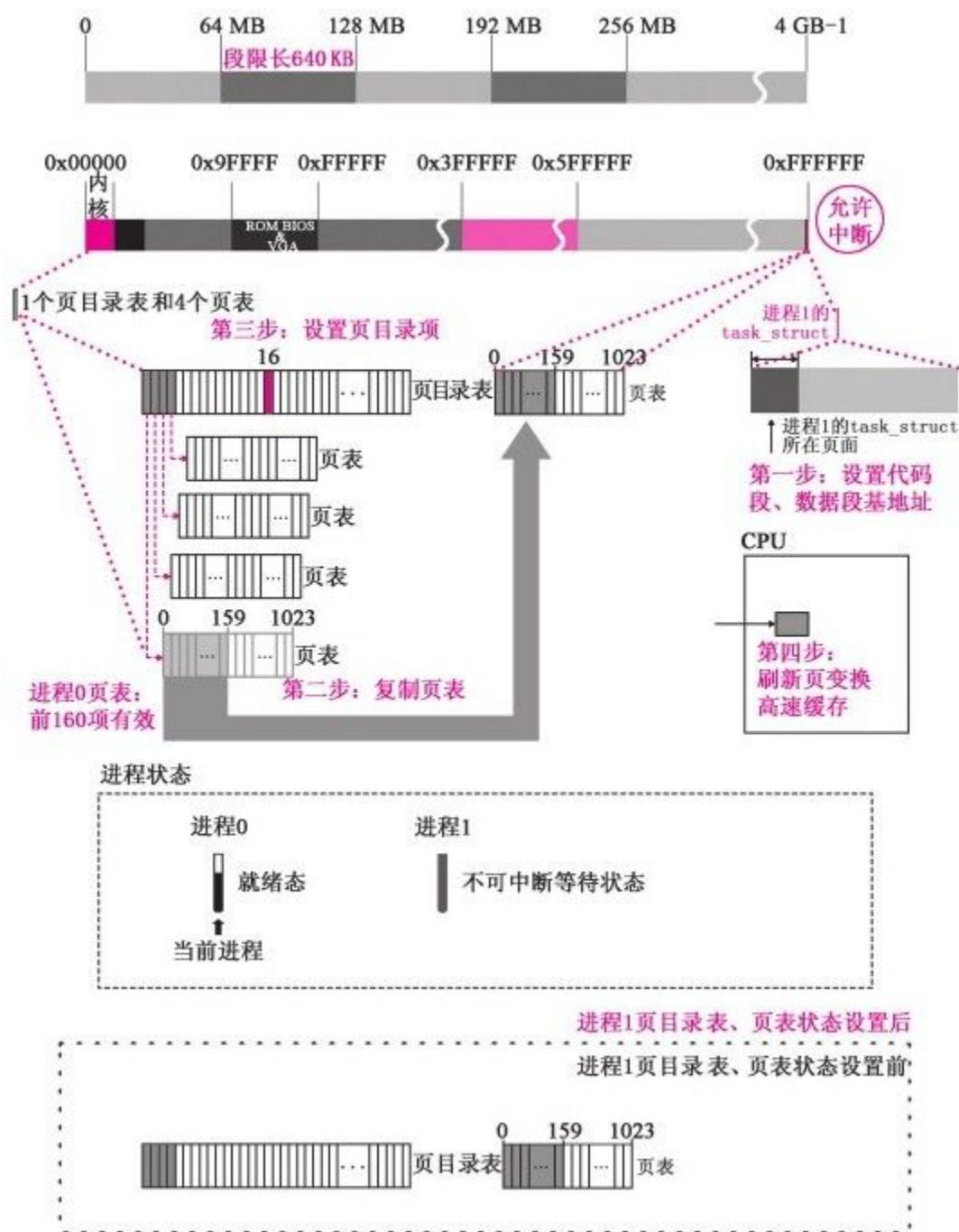


图 3-8 设置进程1的线性地址空间

//代码路径：include/linux/sched.h:

.....

`#define_set_base (addr,base) \`//用base设置addr, 参看2.9节段描述
符图及代码注释

```
__asm__ ("movw%%dx, %0\n\t"
```

```
"rorl$16, %%edx\n\t"
```

```
"movb%%dl, %1\n\t"
```

```
"movb%%dh, %2"
```

```
: "m" (* ((addr) +2) ), \
```

```
"m" (* ((addr) +4) ), \
```

```
"m" (* ((addr) +7) ), \
```

```
"d" (base) \
```

```
: "dx")
```

.....

`#define set_base (ldt,base) _set_base (((char *) & (ldt)) ,`
base)

.....

`#define_get_base (addr) (\`//获取addr段基址, 参看_set_base,
参看2.9节段描述

`//`符图及代码注释

```

unsigned long __base; \

__asm__ ("movb%3, %%dh\n\t\"
"movb%2, %%dl\n\t\"
"shll$16, %%edx\n\t\"
"movw%1, %%dx\"
: "=d" (__base) \
: "m" (* ( (addr) +2) ) , \
"m" (* ( (addr) +4) ) , \
"m" (* ( (addr) +7) ) ) ; \

__base; })

#define get_base (ldt) _get_base ( ( (char *) & (ldt) ) )

#define get_limit (segment) ({\

unsigned long __limit; \

__asm__
("lsl%1, %0\n\tincl%0": "=r" (__limit) : "r" (segment) ) ; \

//取segment的段限长, 给__limit

__limit; })

```

//代码路径: kernel/fork.c:

int copy_mem (int nr,struct task_struct * p) //设置子进程的代码段、数据段及创建、复制子进程的第一个页表

{

unsigned long old_data_base,new_data_base,data_limit;

unsigned long old_code_base,new_code_base,code_limit;

//取子进程的代码、数据段限长

code_limit=get_limit (0x0f) ; //0x0f即1111: 代码段、LDT、3特权级

data_limit=get_limit (0x17) ; //0x17即10111: 数据段、LDT、3特权级

//获取父进程（现在是进程0）的代码段、数据段基址

old_code_base=get_base (current->ldt[1]) ;

old_data_base=get_base (current->ldt[2]) ;

if (old_data_base != old_code_base)

panic ("We don't support separate I&D") ;

if (data_limit < code_limit)

panic ("Bad data_limit") ;

```
new_data_base=new_code_base=nr*0x4000000; //现在nr是1,  
0x4000000是64 MB
```

```
p->start_code=new_code_base;
```

```
set_base (p->ldt[1], new_code_base) ; //设置子进程代码段基  
址
```

```
set_base (p->ldt[2], new_data_base) ; //设置子进程数据段基址
```

```
if (copy_page_tables (old_data_base,new_data_base,data_limit) )  
{
```

```
free_page_tables (new_data_base,data_limit) ;
```

```
return-ENOMEM;
```

```
}
```

```
return 0;
```

```
}
```

2.为进程1创建第一个页表并设置对应的页目录项

在Linux 0.11中，每个进程所属的程序代码执行时，都要根据其线性地址来进行寻址，并最终映射到物理内存上。通过图3-9我们可以看出，线性地址有32位，CPU将这个线性地址解析成“页目录项”、“页表项”和“页内偏移”；页目录项存在于页目录表中，用以管理页表；页表项存在于页表中，用以管理页面，最终在物理内存上找到指定的地址。Linux 0.11中仅有一个页目录表，通过线性地址中提供的“页目录项”数据就可以找到页目录表中对应的页目录项；通过这个页目录项就可以找到对应的页表；之后，通过线性地址中提供的“页表项”数据，就可以在该页表中找到对应的页表项；通过此页表项可以进一步找到对应的物理页面；最后，通过线性地址中提供的“页内偏移”落实到实际的物理地址值。

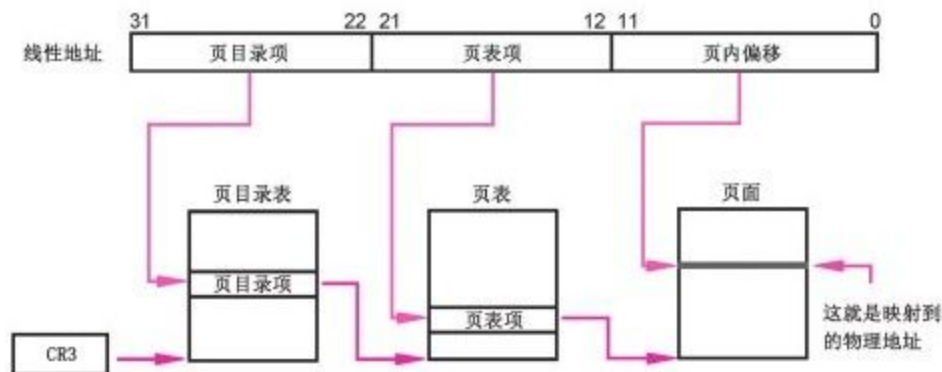


图 3-9 线性地址到物理地址映射过程示意图

调用`copy_page_tables()`函数，设置页目录表和复制页表，如图3-8中第二步和第三步所示，注意其中页目录项的位置。

执行代码如下：

//代码路径：kernel/fork.c:

```
int copy_mem (int nr,struct task_struct * p)
{
    .....
```



```
set_base (p->ldt[1], new_code_base) ; //设置子进程代码段基址  
  
set_base (p->ldt[2], new_data_base) ; //设置子进程数据段基址  
  
//为进程1创建第一个页表、复制进程0的页表，设置进程1的页目录项  
  
if (copy_page_tables (old_data_base,new_data_base,data_limit) )  
{  
  
    free_page_tables (new_data_base,data_limit) ;  
  
    return-ENOMEM;  
  
}  
  
return 0;  
  
}
```

进入copy_page_tables () 函数后，先为新的页表申请一个空闲页面，并把进程0中第一个页表里面前160个页表项复制到这个页面中（1个页表项控制一个页面4 KB内存空间，160个页表项可以控制640 KB内存空间）。进程0和进程1的页表

暂时都指向了相同的页面，意味着进程1也可以操作进程0的页面。之后对进程1的页目录表进行设置。最后，用重置CR3的方法刷新页变换高速缓存。进程1的页表和页目录表设置完毕。

执行代码如下（为了更容易读懂，我们在源代码中做了比较详细的注释）：

```
//代码路径: mm/memory.c:

.....

#define invalidate () \

__asm__ ("movl%%eax, %%cr3": "a" (0) ) //重置CR3为0

.....

int copy_page_tables (unsigned long from,unsigned long to,long
size)
{
    unsigned long * from_page_table;
```

```

unsigned long * to_page_table;

unsigned long this_page;

unsigned long * from_dir, *to_dir;

unsigned long nr;

```

/*0x3ffff是4 MB，是一个页表的管辖范围，二进制是22个1，||的两边必须同为0，所以，from和to后22位必须都为0，即4 MB的整数倍，意思是一个页表对应4 MB连续的线性地址空间必须是从0x000000开始的4 MB的整数倍的线性地址，不能是任意地址开始的4 MB，才符合分页的要求*/

```

if ( (from&0x3ffff) || (to&0x3ffff) )

```

```

panic ("copy_page_tables called with wrong alignment") ;

```

/*一个页目录项的管理范围是4 MB，一项是4字节，项的地址就是项数×4，也就是项管理的线性地址起始地址的M数，比如：0项的地址是0，管理范围是0~4 MB，1项的地址是4，管理范围是4~8 MB，2项的地址是8，管理范围是8~12MB.....>>20就是地址的MB数，&0xffc就是&111111111100b，就是4 MB以下部分清零的地址的MB数，也就是页目录项的地址*/

```

from_dir= (unsigned long *) ( (from>>20) &
0xffc) ; /*_pg_dir=0*/

```

```

to_dir= (unsigned long *) ( (to>>20) &0xffc) ;

```

数
size= ((unsigned) (size+0x3ffff)) >>22; //>>22是4 MB

```

for (; size-->0; from_dir++, to_dir++) {

```

```
if (1&*to_dir)
```

```
panic ("copy_page_tables: already exist") ;
```

```
if (! (1&*from_dir) )
```

```
continue;
```

/*from_dir是页目录项中的地址，0xfffff000&是将低12位清零，高20位是页表的地址

```
from_page_table= (unsigned long *) (0xfffff000&*from_dir) ;
```

```
if (! (to_page_table= (unsigned long *) get_free_page () ) )
```

```
return-1; /*Out of memory,see freeing*/
```

*to_dir= ((unsigned long) to_page_table) |7; //7即111，参看1.3.5节的注释

```
nr= (from==0) ?0xA0: 1024; //0xA0即160，复制页表的项数，
```

for (; nr-->0; from_page_table++, to_page_table++) {//复制父进程页表

```
this_page=*from_page_table;
```

```
if (! (1&this_page) )
```

```
continue;
```

this_page&=~2; //设置页表项属性，2是010，~2是101，代表用户、只读、存在

```
*to_page_table=this_page;

if (this_page > LOW_MEM) { //1 MB以内的内核区不参与用户分
页管理

*from_page_table=this_page;

this_page-=LOW_MEM;

this_page >>=12;

mem_map[this_page]++; //增加引用计数，参看mem_init

}

}

}

invalidate ( ) ; //用重置CR3为0，刷新"页变换高速缓存"

return 0;

}
```

进程1此时是一个空架子，还没有对应的程序，它的页表又是从进程0的页表复制过来的，它们管理的页面完全一致，也就是它暂时和进程0共

享一套内存页面管理结构，如图3-10所示。等将来它有了自己的程序，再把关系解除，并重新组织自己的内存管理结构。

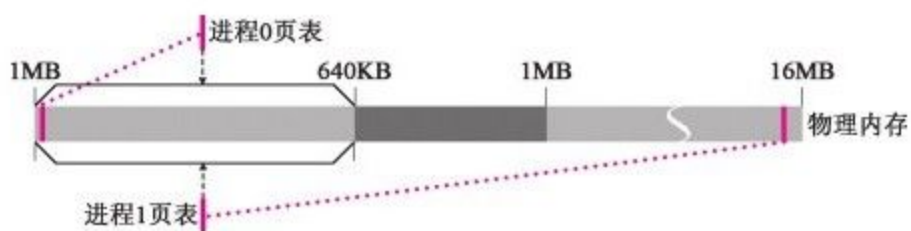


图 3-10 进程0和进程1共享页表示意图

3.1.5 进程1共享进程0的文件

返回`copy_process()`函数中继续调整。设置`task_struct`中与文件相关的成员，包括打开了哪些文件`p->filp[20]`、进程0的“当前工作目录i节点结构”、“根目录i节点结构”以及“执行文件i节点结构”。虽然进程0中这些数值还都是空的，进程0只具备在主机中正常运算的能力，尚不具备与外设以文件形式进行交互的能力，但这种共享仍有意义，因为父子进程创建机制会把这种能力“遗传”给子进程。

对应的代码如下：

```
//代码路径: kernel/fork.c:
```

```
int copy_process (int nr,long ebp,long edi,long esi,long gs,long  
none,
```

```
long ebx,long ecx,long edx,
```

```
long fs,long es,long ds,
```

```
long eip,long cs,long eflags,long esp,long ss)
```

```
{
```

```
.....
```

```
return-EAGAIN;
```

```
}
```

for (i=0; i<NR_OPEN; i++) //下面将父进程相关文件属性的引用计数加1，表明父子进程共享文件

```
if (f=p->filp[i])
```

```
f->f_count++;
```

```
if (current->pwd)
```

```
current->pwd->i_count++;
```

```
if (current->root)
```

```
current->root->i_count++;
```

```
if (current->executable)
```



```
current->executable->i_count++;
```

```
    set_tss_desc (gdt+ (nr<<1) +FIRST_TSS_ENTRY, & (p->tss) ) ; //设置GDT中与子进程相关的项, 参看sched.c
```

```
.....
```

```
}
```

3.1.6 设置进程1在GDT中的表项

之后把进程1的TSS和LDT，挂接在GDT中，
如图3-11所示，注意进程1在GDT中所占的位置。

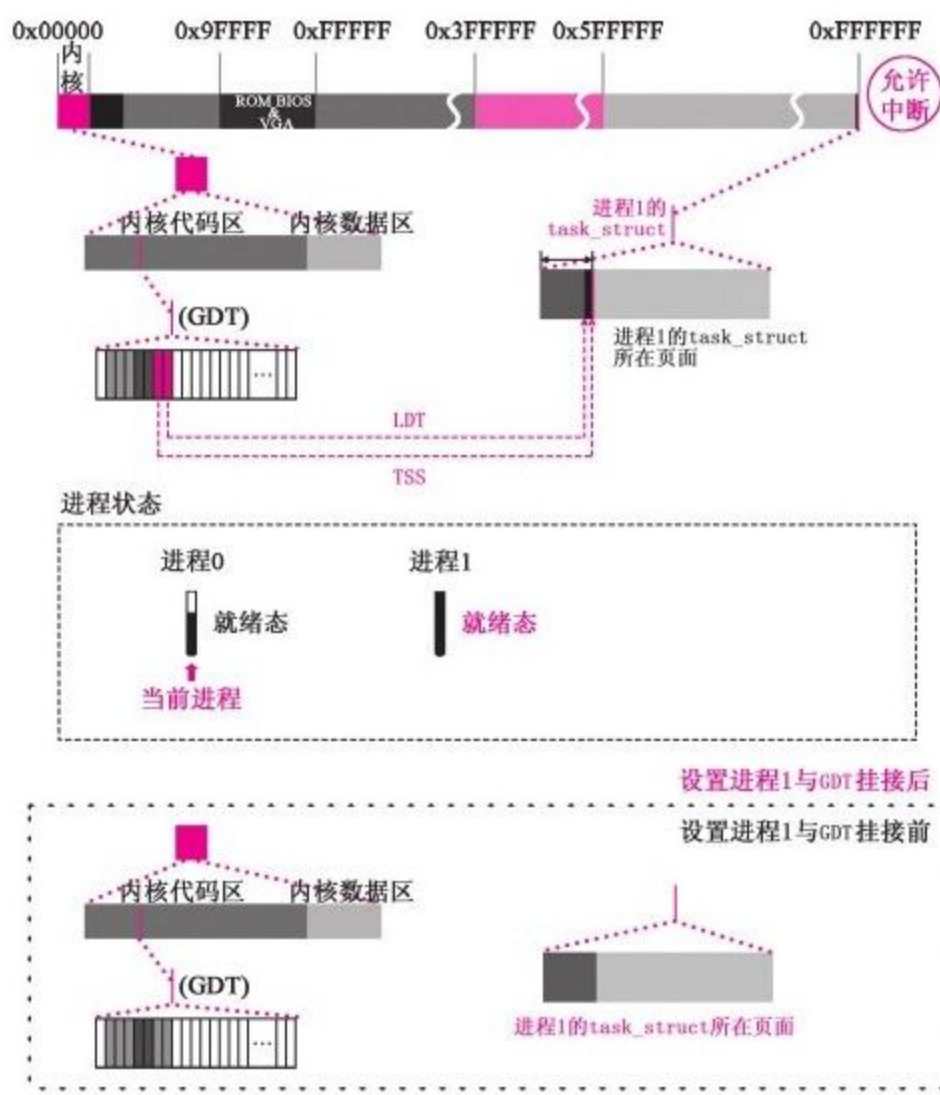


图 3-11 将进程1的task_struct与GDT挂接

执行代码如下：

//代码路径: kernel/fork.c:

```
int copy_process (int nr,long ebp,long edi,long esi,long gs,long
none,

long ebx,long ecx,long edx,

long fs,long es,long ds,

long eip,long cs,long eflags,long esp,long ss)

{

.....

current->executable->i_count++;

set_tss_desc (gdt+ (nr<<1) +FIRST_TSS_ENTRY, & (p->
tss) ) ; //设置GDT中与子进程相关的项, 参看sched.c

set_ldt_desc (gdt+ (nr<<1) +FIRST_LDT_ENTRY, & (p->
ldt) ) ;

p->state=TASK_RUNNING; /*do this last,just in case*///设置子进
程为就绪态
```

.....}

3.1.7 进程1处于就绪态

将进程1的状态设置为就绪态，使它可以参加进程调度，最后返回进程号1。请注意图3-11中间代表进程的进程条，其中，进程1已处在就绪态。执行代码如下：

//代码路径： kernel/fork.c:

```
int copy_process (int nr,long ebp,long edi,long esi,long gs,long
none,

long ebx,long ecx,long edx,

long fs,long es,long ds,

long eip,long cs,long eflags,long esp,long ss)

{

.....

p->state=TASK_RUNNING; /*do this last,just in case*///设置子进
程为就绪态
```

```
return last_pid;  
  
}
```

至此，进程1的创建工作完成，进程1已经具备了进程0的全部能力，可以在主机中正常地运行。

进程1创建完毕后，`copy_process()` 函数执行完毕，返回`sys_fork()` 中`call_copy_process()` 的下一行执行，执行代码如下：

```
//代码路径： kernel/system_call.s:
```

```
.....
```

```
_sys_fork:
```

```
call _find_empty_process
```

```
testl %eax, %eax
```

```
js 1f
```

```
push %gs
```

```
pushl %esi
```

```
pushl %edi
```

```
pushl %ebp
```

```
pushl %eax
```

```
call _copy_process
```

addl \$20, %esp//copy_process返回至此，esp+=20就是esp清20字节的栈，也就是清前面压的gs、esi、

1: ret//edi、ebp、eax，注意：内核栈里还有数据。返回
_system_call中的pushl%eax执行

.....

清_sys_fork压栈的5个寄存器的值，就是清前面压的gs、esi、edi、ebp、eax，也就是
copy_process（）的前5个参数。注意：eax对应的是copy_process（）的第一个参数nr，就是
copy_process（）的返回值last_pid，即进程1的进

程号。然后返回_system_call中的
call_sys_call_table (, %eax, 4) 的下一行
pushl%eax处继续执行。

先检查当前进程是否是进程0。注意：
pushl%eax这行代码，将3.1.6节中返回的进程1的
进程号压栈，之后到_ret_from_sys_call: 处执
行。

执行代码如下：

```
//代码路径: kernel/system_call.s:
```

```
.....
```

```
_system_call:
```

```
.....
```

```
call _sys_call_table (, %eax, 4)
```


pushl %eax#sys_fork返回到此执行，eax是copy_process（）的返回值last_pid

movl _current, %eax#当前进程是进程0

cmpl \$0, state (%eax) #state

jne reschedule#如果进程0不是就绪态，则进程调度

cmpl \$0, counter (%eax) #counter

je reschedule#如果进程0没有时间片，则进程调度

ret _from_sys_call:

movl _current, %eax#task[0]cannot have signals

cmpl _task, %eax

je 3f#如果当前进程是进程0，跳到下面的3: 处执行。当前进程是进程0！

cmpw \$0x0f,CS (%esp) #was old code segment supervisor?

jne 3f

cmpw \$0x17, OLDSS (%esp) #was stack segment=0x17?

jne 3f

movl signal (%eax) , %ebx

movl blocked (%eax) , %ecx

notl %ecx

andl %ebx, %ecx

bsfl %ecx, %ecx

je 3f

btrl %ecx, %ebx

movl %ebx,signal (%eax)

incl %ecx

pushl %ecx

call _do_signal

popl %eax

3: popl%eax#如果是进程0，则直接跳到这个地方执行，将7个寄存器的值出栈给CPU

popl %ebx

popl %ecx

popl %edx

pop %fs

pop %es

```
pop %ds
```

```
iret
```

#CPU硬件将int 0x80的中断时压的ss、esp、eflags、cs、eip的值出栈给CPU对应寄存器，CS: EIP指向fork () 中int 0x80的下一行if (__res >= 0) 处执行

```
.....
```

由于当前进程是进程0，所以就跳转到标号3处，将压栈的各个寄存器数值还原。图3-12表示了init_task中清栈的这一过程。值得注意的是popl%eax这一行代码，这是将前面刚刚讲解过的pushl%eax压栈的进程1的进程号，恢复给CPU的eax, eax的值为“1”。

之后，iret中断返回，CPU硬件自动将int 0x80的中断时压的ss、esp、eflags、cs、eip的值按压栈的反序出栈给CPU对应寄存器，从0特权级的

内核代码转换到3特权级的进程0代码执行，CS:
EIP指向fork () 中int 0x80的下一行if (__res >=0) 。

对应的执行代码如下：

```
//代码路径: include/unistd.h:

int fork (void)

{

    long __res;

    __asm__volatile ("int$0x80"

        : "=a" (__res) //__res的值就是eax, 是copy_process () 的返回值
last_pid (1)

        : "0" (__NR_fork) ) ;

    if (__res >=0) //iret后, 执行这一行! __res就是eax, 值是1

    return (int) __res; //返回1!

    errno=-__res;
```

```
return-1;
```

```
}
```

在讲述执行if (__res >=0) 前，先关注一下： "=a" (__res) 。这行代码的意思是将__res的值赋给eax，所以if (__res >=0) 这一行代码，实际上就是判断此时eax的值是多少。我们刚刚介绍了，这时候eax里面的值是返回的进程1的进程号1，return (type) __res将“1”返回。回到3.1.1节中fork () 函数的调用点if (! fork ()) 处执行，！ 1为“假”，这样就不会执行到init () 函数中，而是进程0继续执行，接下来就会执行到for (;) pause () 。

执行代码如下：

```
//代码路径: init/main.c:
```

```
.....

void main (void)

{

sti ( ) ;

move _to_user_mode ( ) ;

if ( ! fork ( ) ) { //fork的返回值为1, if ( ! 1) 为假/*we count on
this going ok*/

init ( ) ; //不会执行这一行

}

.....

for ( ; ) pause ( ) ; //执行这一行!

}
```

图3-12形象地表示了上述过程。

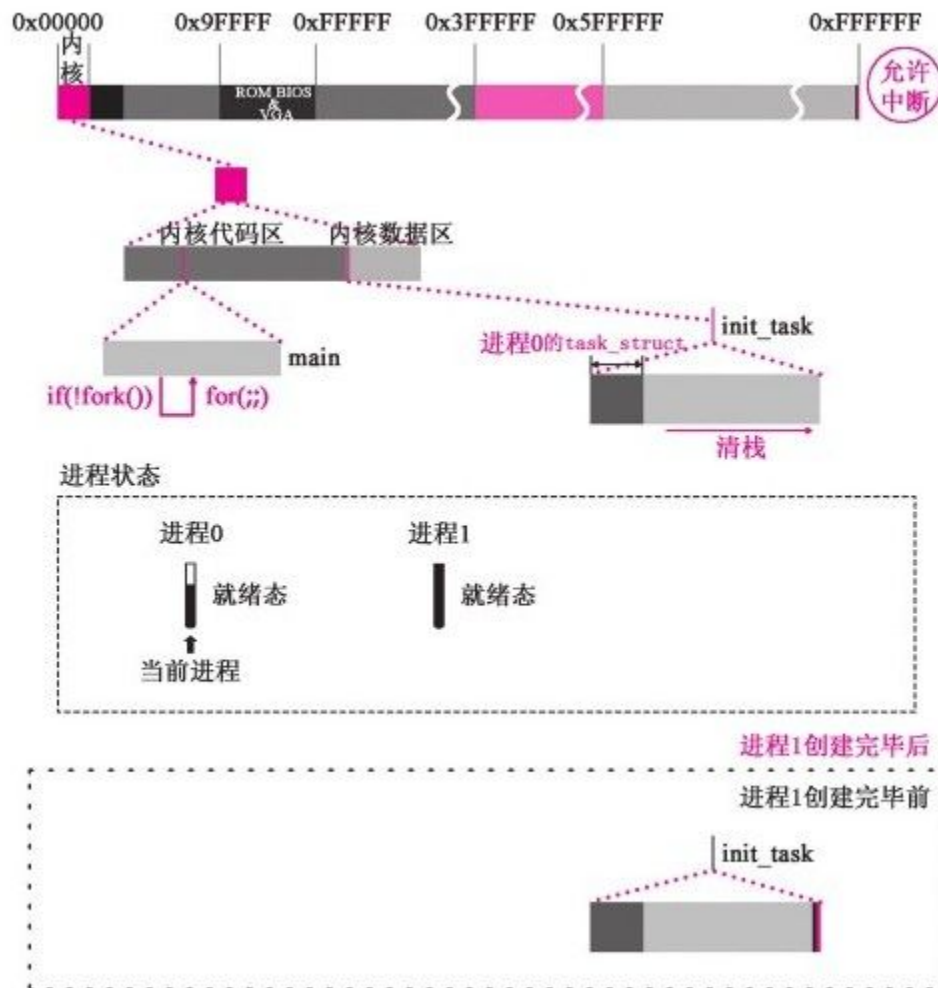


图 3-12 操作系统如何区分进程0与进程1

3.2 内核第一次做进程调度

现在执行的是进程0的代码。从这里开始，进程0准备切换到进程1去执行。

在Linux 0.11的进程调度机制中，通常有以下两种情况可以产生进程切换。

1) 允许进程运行的时间结束。

进程在创建时，都被赋予了有限的时间片，以保证所有进程每次都只执行有限的时间。一旦进程的时间片被削减为0，就说明这个进程此次执行的时间用完了，立即切换到其他进程去执行，实现多进程轮流执行。

2) 进程的运行停止。

当一个进程需要等待外设提供的数据，或等待其他程序的运行结果.....或进程已经执行完毕时，在这些情况下，虽然还有剩余的时间片，但是进程不再具备进一步执行的“逻辑条件”了。如果还等着时钟中断产生后再切换到别的进程去执行，就是在浪费时间，应立即切换到其他进程去执行。

这两种情况中任何一种情况出现，都会导致进程切换。

进程0角色特殊。现在进程0切换到进程1既有第二种情况的意思，又有怠速进程的意思。我们会在3.3.1节中讲解怠速进程。

进程0执行for (;) pause () ，最终执行到 schedule () 函数切换到进程1，如图3-13所示。

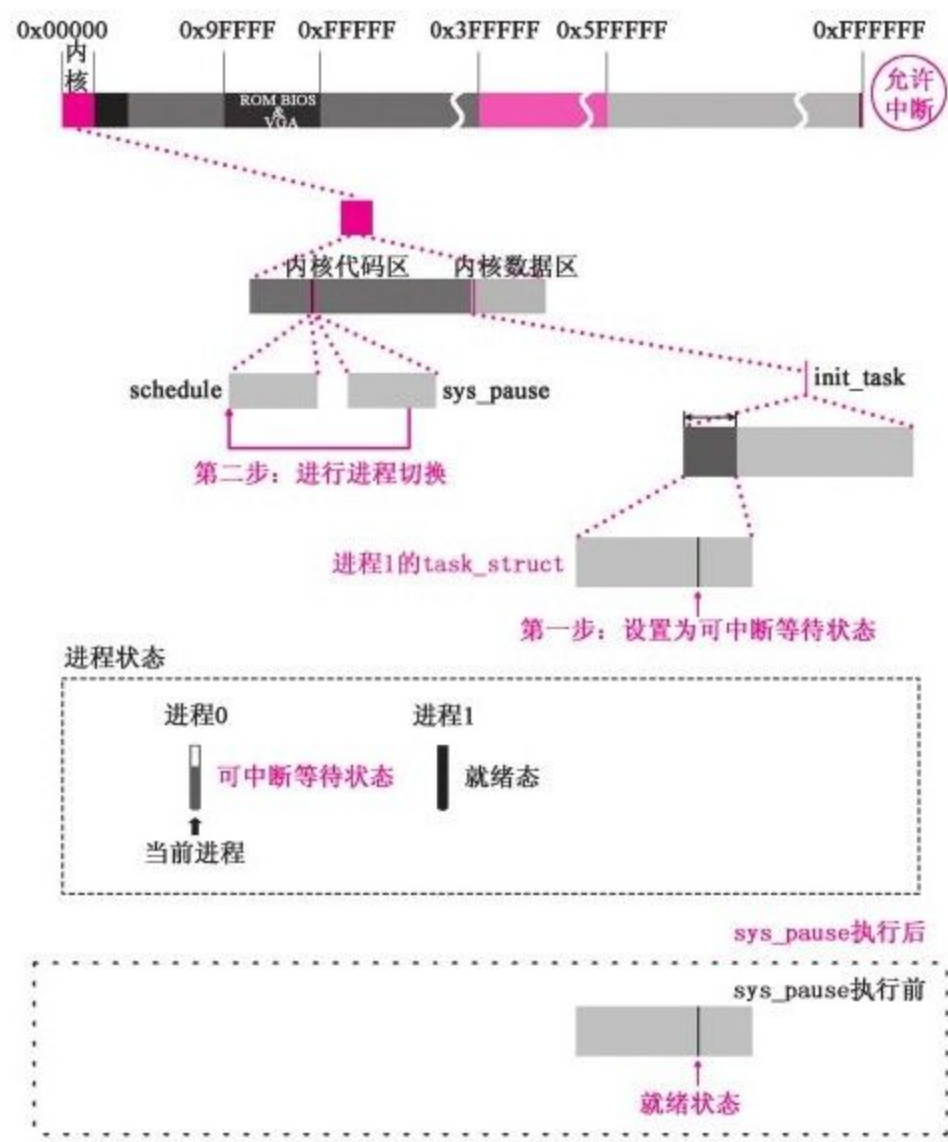


图 3-13 进程0挂起并执行调度程序

pause函数的执行代码如下：

```
//代码路径： init/main.c:
```

```
.....
```

```
static inline_syscall0 (int,fork)
```

```
static inline_syscall0 (int,pause)
```

```
.....
```

```
void main (void)
```

```
{
```

```
.....
```

```
move _to_user_mode () ;
```

```
if (! fork () ) { /*we count on this going ok*/
```

```
init () ;
```

```
}
```

```
for (; ) pause () ;
```

```
}
```

`pause ()` 函数的调用与 `fork ()` 函数的调用一样，会执行到 `unistd.h` 中的 `syscall0`，通过 `int 0x80` 中断，在 `system_call.s` 中的 `call_sys_call_table (, %eax, 4)` 映射到 `sys_pause ()` 的系统调用函数去执行，具体步骤与 3.1.1 节中调用 `fork ()` 函数步骤类似。略有差别的是，`fork ()` 函数是用汇编写的，而 `sys_pause ()` 函数是用 C 语言写的。

进入 `sys_pause ()` 函数后，将进程 0 设置为可中断等待状态，如图 3-13 中第一步所示，然后调用 `schedule ()` 函数进行进程切换，执行代码如下：

```
//代码路径： kernel/sched.c:
```

```
int sys_pause (void)
```

```
{  
  
    //将进程0设置为可中断等待状态，如果产生某种中断，或其他进程给这个进程发送特定信号.....才有可能将//这个进程的状态改为就绪态  
  
    current->state=TASK_INTERRUPTIBLE;  
  
    schedule ();  
  
    return 0;  
  
}
```

在schedule () 函数中，先分析当前有没有必要进行进程切换，如果有必要，再进行具体的切换操作。

首先依据task[64]这个结构，第一次遍历所有进程，只要地址指针不为空，就要针对它们的“报警定时值alarm”以及“信号位图signal”进行处理

（我们会在后续章节详细讲解信号，这里先不深究）。在当前的情况下，这些处理还不会产生具

体的效果，尤其是进程0此时并没有收到任何信号，它的状态是“可中断等待状态”，不可能转变为“就绪态”。

第二次遍历所有进程，比较进程的状态和时间片，找出处在就绪态且counter最大的进程。现在只有进程0和进程1，且进程0是可中断等待状态，不是就绪态，只有进程1处于就绪态，所以，执行switch_to (next)，切换到进程1去执行，如图3-14中的第一步所示。

执行代码如下：

```
//代码路径： kernel/sched.c:
```

```
void schedule (void)
```

```
{
```

```
int i,next,c;
```

```

struct task_struct ** p;

/*check alarm,wake up any interruptible tasks that have got a signal*/

for (p=&LAST_TASK; p> &FIRST_TASK; --p)

if (*p) {

    if ( (*p) -> alarm&& (*p) -> alarm<jiffies) {//如果设置了定时或定时已过

        (*p) -> signal|= (1<< (SIGALRM-1)) ; //设置SIGALRM

        (*p) -> alarm=0; //alarm清零

    }

    if ( ( (*p) -> signal&~ (_BLOCKABLE& (*p) ->
blocked) ) &&

        (*p) -> state==TASK_INTERRUPTIBLE) //现在还不是这种情况

        (*p) -> state=TASK_RUNNING;

    }

/*this is the scheduler proper: */

while (1) {

c=-1;

next=0;

```

```
i=NR_TASKS;
```

```
p=&task[NR_TASKS];
```

```
while (--i) {
```

```
if (! *--p)
```

```
continue;
```

```
if ( (*p) -> state==TASK_RUNNING && (*p) -> counter >
```

c) //找出就绪态中counter最大的进程

```
c= (*p) -> counter,next=i;
```

```
}
```

```
if (c) break;
```

```
for (p=&LAST_TASK; p > &FIRST_TASK; --p)
```

```
if (*p)
```

```
    (*p) -> counter= ( (*p) -> counter > > 1) +
```

```
    (*p) -> priority; //即counter=counter/2+priority
```

```
}
```

```
switch_to (next) ;
```

```
}
```


//代码路径: include/sched.h:

.....

//FIRST_TSS_ENTRY << 3是100000, ((unsigned long) n) <
< 4, 对进程1是10000

//_TSS (1) 就是110000, 最后2位特权级, 左第3位GDT, 110是6
即GDT中tss0的下标

#define_TSS (n) ((((unsigned long) n) << 4) +
(FIRST_TSS_ENTRY << 3))

.....

#define switch_to (n) {V/参看2.9.1节

struct{long a,b; }__tmp; V/为ljmp的CS、EIP准备的数据结构

__asm__ ("cmpl%%ecx, _current\n\t\"

"je 1f\n\t"V/如果进程n是当前进程, 没必要切换, 退出

"movw%%dx, %1\n\t"V/EDX的低字赋给*&__tmp.b, 即把CS赋
给.b

"xchgl%%ecx, _current\n\t"V/task[n]与task[current]交换

"ljmp%0\n\t"V/ljmp到__tmp, __tmp中有偏移、段选择符, 但任务
门忽略偏移

"cmpl%%ecx, _last_task_used_math\n\t"V/比较上次是否使用过协
处理器

```

"jne 1f\n\t"

"clts\n"\\清除CR0中的切换任务标志

"1:  \"

: "m" (*&__tmp.a) , "m" (*&__tmp.b) , \/.a对应EIP (忽略) , .b对应CS

"d" (_TSS (n) ) , "c" ( (long) task[n] ) ) ; \/.EDX是TSS n的索引号, ECX即task[n]

}

```

程序将一直执行到"ljmp%0\n\t"这一行。ljmp通过CPU的任务门机制并未实际使用任务门，将CPU的各个寄存器值保存在进程0的TSS中，将进程1的TSS数据以及LDT的代码段、数据段描述符数据恢复给CPU的各个寄存器，实现从0特权级的内核代码切换到3特权级的进程1代码执行，如图3-14中的第二步所示。

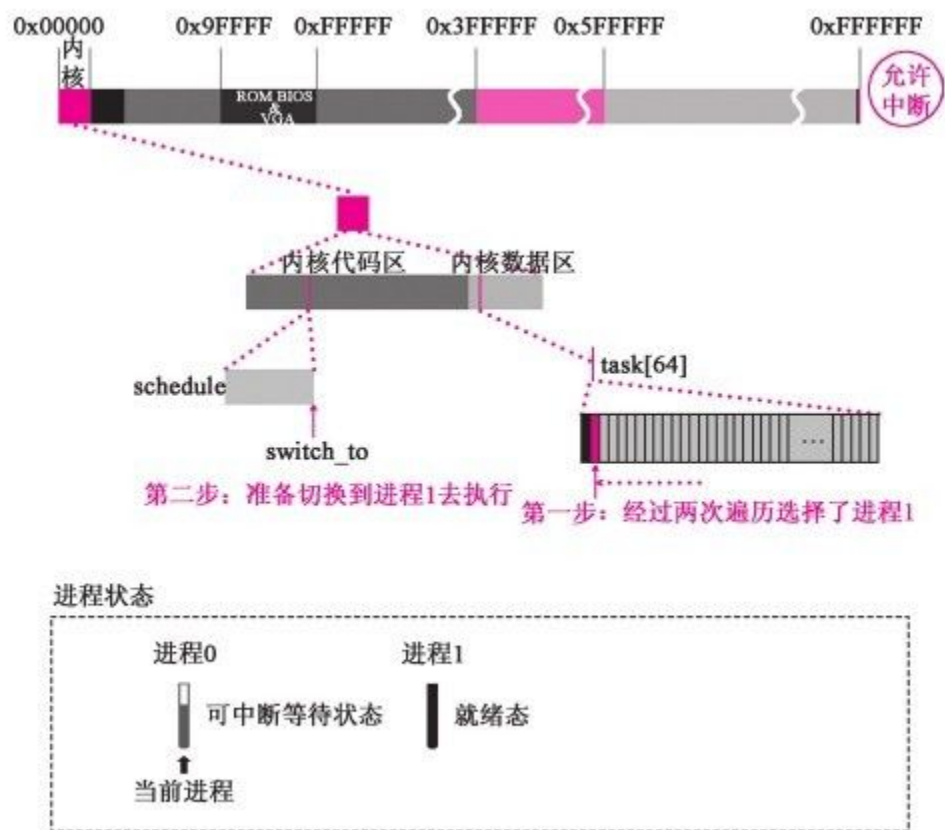


图 3-14 调度进程1执行

接下来，轮到进程1执行，它将进一步构建环境，使进程能够以文件的形式与外设交互。

需要提醒的是，`pause()` 函数的调用是通过 `int 0x80` 中断从3特权级的进程0代码翻转到0特权级的内核代码执行的，在 `_system_call` 中的

call_sys_call_table (, %eax, 4) 中调用sys_pause
() 函数, 并在sys_pause () 中的schedule () 中
调用switch () , 在switch () 中ljmp进程1的代码
执行。现在, switch () 中ljmp后面的代码还没有
执行, call_sys_call_table (, %eax, 4) 后续的代码
也还没有执行, int 0x80的中断没有返回。

3.3 轮转到进程1执行

在分析进程1如何开始执行之前，先回顾一下进程0创建进程1的过程。

在3.1.3节中讲解调用`copy_process`函数时曾强调过，当时为进程1设置的`tss.eip`就是进程0调用`fork()`创建进程1时`int 0x80`中断导致的CPU硬件自动压栈的`ss`、`esp`、`eflags`、`cs`、`eip`中的EIP值，这个值指向的是`int 0x80`的下一行代码的位置，即`if (__res >= 0)`。

前面讲述的`ljmp`通过CPU的任务门机制自动将进程1的TSS的值恢复给CPU，自然也将其中的`tss.eip`恢复给CPU。现在CPU中的EIP指向的就是

fork中的if (__res >=0) 这一行，所以，进程1就要从这一行开始执行。

执行代码如下：

```
//代码路径: include/unistd.h:

#define _syscall0 (type,name) \

int fork (void)

{

long __res;

__asm__volatile ("int$0x80"

: "=a" (__res)

: "0" (__NR_fork) );

if (__res >=0) //现在从这行开始执行， copy_process为进程1做的
tss.eip就是指向这一行

return (int) __res;

errno=-__res;
```

```
return-1;

}
```

回顾前面3.1.3节中的介绍可知，此时的__res值，就是进程1的TSS中eax的值，这个值在3.1.3节中被写死为0，即p->tss.eax=0，因此，当执行到return (type) __res这一行时，返回值是0，如图3-15所示。

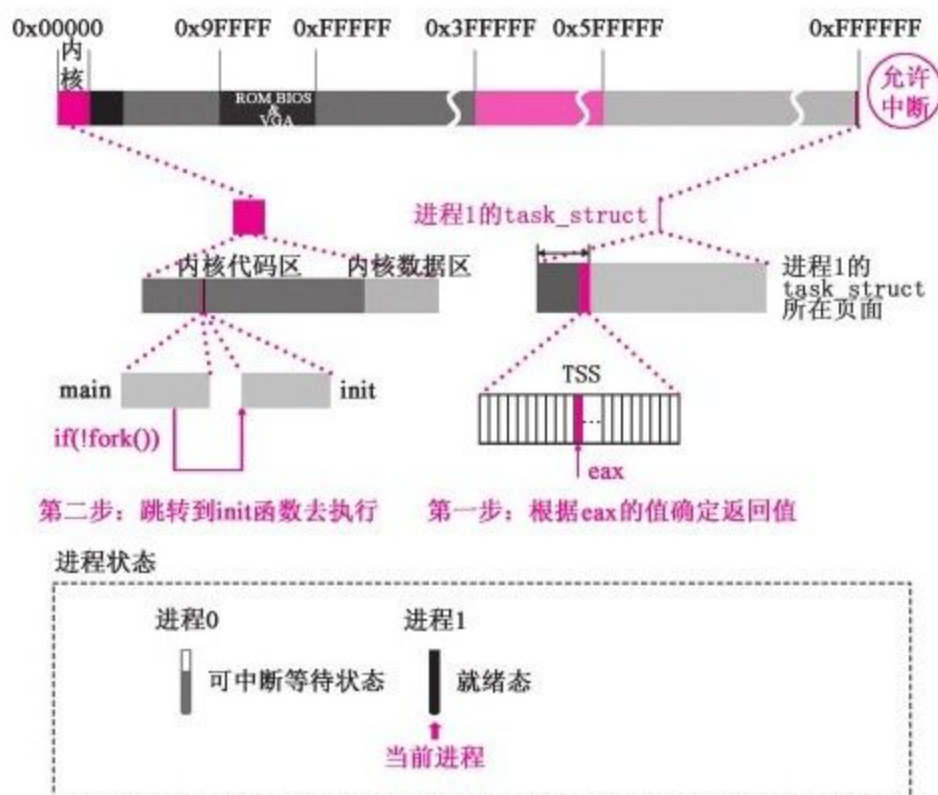


图 3-15 进程1开始执行的状态

返回后，执行到main（）函数中if（！ fork（））这一行，！ 0为“真”，调用init（）函数！
执行代码如下：

```
//代码路径： init/main.c:
```

```
void main（void）
```

```
{
```

```
.....
```

```
if（！ fork（））{ //！ 0为真，
```

```
init（）； //这次要执行这一行！ 代码跨度比较大， 请参看3.1.3节
```

```
}
```

```
}
```

进入init（）函数后，先调用setup（）函数，
执行代码如下：

```
//代码路径： init/main.c:

void init（void）

{

.....

setup（（void*）&drive_info）；

.....

}
```

本章后续的内容都是setup（）函数实现的。
这个函数的调用与fork（）、pause（）函数的调用类似；略有区别的是setup（）函数不是通过_syscall0（）而是通过_syscall1（）实现的；具体的实现过程基本类似，也是通过int 0x80、

`_system_call`、`call_sys_call_table`（, `%eax`, 4）、`sys_setup`（）。

提醒：前面`pause`（）函数的那个`int 0x80`中断还没有返回，现在`setup`（）又产生了一个中断。

3.3.1 进程1为安装硬盘文件系统做准备

这一节的内容涉及`sys_setup`（）的大部分代码，包括从函数开始到调用`rd_load`（）之前的所有代码；技术路线比较长，代码很多，难度比较大，`hash_table`的部分尤其如此。但这部分代码的目的却很单一：为第5章将要讲述的安装硬盘文件系统做准备。

这个过程大概经过3个步骤；

- 1) 根据机器系统数据设置硬盘参数;
- 2) 读取硬盘引导块;
- 3) 从引导块中获取信息。

1.进程1设置硬盘的hd_info

根据机器系统数据中的drive_info，如硬盘的柱面数、磁头数、扇区数，设置内核的hd_info，如图3-16所示。

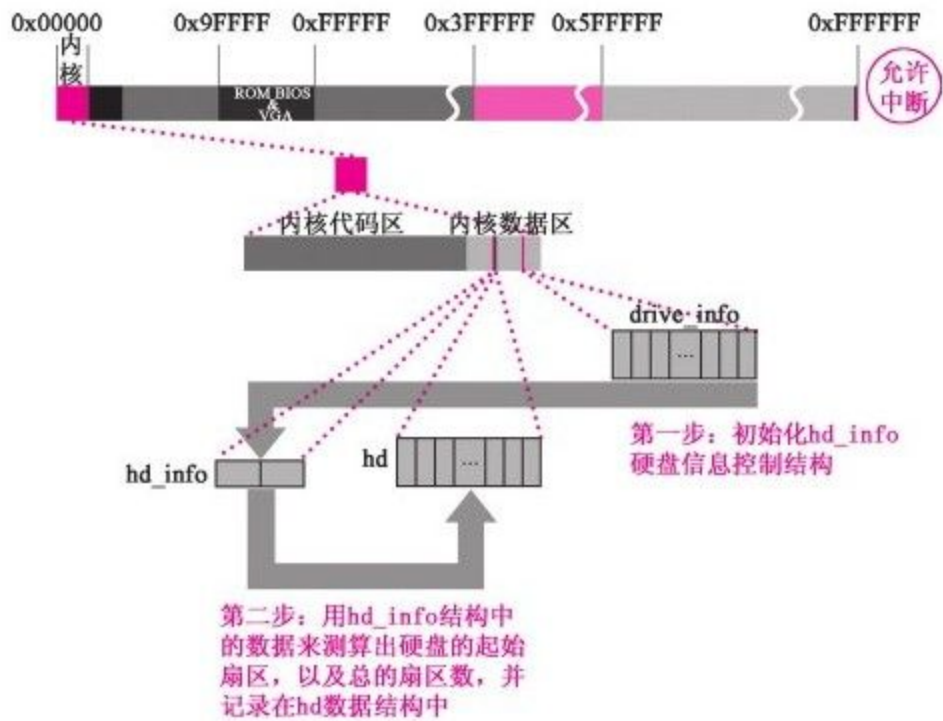


图 3-16 初始化硬盘控制数据结构

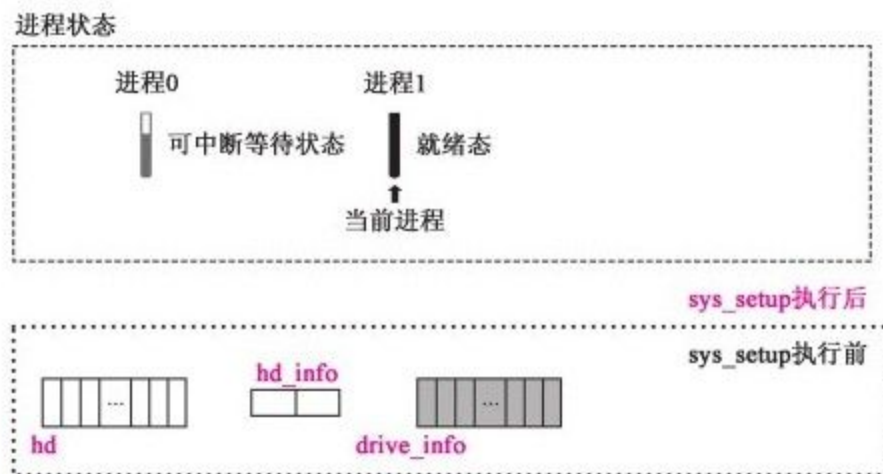


图 3-16 (续)

具体的执行代码如下：

//代码路径: kernel/blk_dev/hd.c:

.....

```
struct hd_i_struct{
```

```
int head,sect,cyl,wpcml,lzone,ctl;
```

```
};
```

.....

```
struct hd_i_struct hd_info[]={ {0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0},  
0, 0}};
```

.....

```
static struct hd_struct{
```

```
long start_sect; //起始扇区号
```

```
long nr_sects; //总扇区数
```

```
}hd[5*MAX_HD]={ {0, 0}, };
```

.....

```
/*This may be used only once,enforced by'static int callable'*/
```

```
int sys_setup (void * BIOS) //对比调用可以看出BIOS就是  
drive_info, 参看2.1节
```

```
{

static int callable=1;

int i,drive;

unsigned char cmos_disks;

struct partition * p;

struct buffer_head * bh;

if (! callable) //控制只调用一次

return-1;

callable=0;

#ifdef HD_TYPE

for (drive=0; drive<2; drive++) { //读取drive_info设置hd_info

hd_info[drive].cyl=* (unsigned short *) BIOS; //柱面数

hd_info[drive].head=* (unsigned char *) (2+BIOS); //磁头数

hd_info[drive].wpcom=* (unsigned short *) (5+BIOS);

hd_info[drive].ctl=* (unsigned char *) (8+BIOS);

hd_info[drive].lzone=* (unsigned short *) (12+BIOS);
```

```
    hd_info[drive].sect=* (unsigned char *) (14+BIOS) ; //每磁道  
扇区数
```

```
    BIOS+=16;
```

```
}
```

```
if (hd_info[1].cyl) //判断有几个硬盘
```

```
NR_HD=2;
```

```
else
```

```
NR_HD=1;
```

```
#endif
```

```
//一个物理硬盘最多可以分4个逻辑盘，0是物理盘，1~4是逻辑  
盘，共5个，第1个物理盘是0*5，第2个物理盘是1*5
```

```
for (i=0; i<NR_HD; i++) {
```

```
    hd[i*5].start_sect=0;
```

```
    hd[i*5].nr_sects=hd_info[i].head*
```

```
    hd_info[i].sect * hd_info[i].cyl;
```

```
}
```

```
if ( (cmos_disks=CMOS_READ (0x12) ) & 0xf0)
```

```
if (cmos_disks & 0x0f)
```

```
NR_HD=2;
```

```
else
```

```
NR_HD=1;
```

```
else
```

```
NR_HD=0;
```

```
for (i=NR_HD; i<2; i++) {
```

```
    hd[i*5].start_sect=0;
```

```
    hd[i*5].nr_sects=0;
```

```
}
```

//第1个物理盘设备号是0x300，第2个是0x305，读每个物理硬盘的0号块，即引导块，有分区信息

```
for (drive=0; drive<NR_HD; drive++) {
```

```
    if (! (bh=bread (0x300+drive*5, 0) ) ) { //
```

```
        printk ("Unable to read partition table of drive%d\n\r",
```

```
        drive) ;
```

```
        panic ("") ;
```

```
}
```


.....

}

2. 读取硬盘的引导块到缓冲区

在Linux 0.11中，硬盘最基础的信息就是分区表，其他信息都可以从这个信息引导出来，这个信息所在的块就是引导块。一块硬盘只有唯一的一个引导块，即硬盘的0号逻辑块。引导块有两个扇区，真正有用的是第一个扇区。我们设定计算机只有一块硬盘。下面把硬盘的引导块读入缓冲区，以便后续程序解读引导块中的信息。这个工作通过调用**bread ()** 函数实现，**bread ()** 可以理解为**block read**。

执行代码如下：

//代码路径: kernel/blk_dev/hd.c:

.....

//第1个物理盘设备号是0x300, 第2个是0x305, 读每个物理硬盘的0号块, 即引导块, 有分区信息

```
for (drive=0; drive<NR_HD; drive++) {  
  
    if (! (bh=bread (0x300+drive*5, 0) ) ) {  
  
        printk ("Unable to read partition table of drive%d\n\r",  
  
drive) ;  
  
        panic ("") ;  
  
    }  
  
    .....  
  
}
```

进入bread () 函数后, 先调用getblk () 函数, 在缓冲区中申请一个空闲的缓冲块。

执行代码如下:

//代码路径: fs/buffer.c:

struct buffer_head * bread (int dev,int block) //读指定dev、block,
第一块硬盘的dev是0x300, block是0

{

struct buffer_head * bh;

if (! (bh=getblk (dev,block))) //在缓冲区得到与dev、block
相符合或空闲的缓冲块

panic ("bread: getblk returned NULL\n"); //现在第一次使用缓冲
区, 不可能没有空闲块

if (bh->b_uptodate) //现在是第一次使用, 找到的肯定是未被使
用过的

return bh;

ll_rw_block (READ,bh) ;

wait_on_buffer (bh) ;

if (bh->b_uptodate)

return bh;

brelse (bh) ;

return NULL;

}

申请空闲缓冲块的主要步骤，在图3-17中已有形象的说明。以下将结合代码来深入分析这一过程。

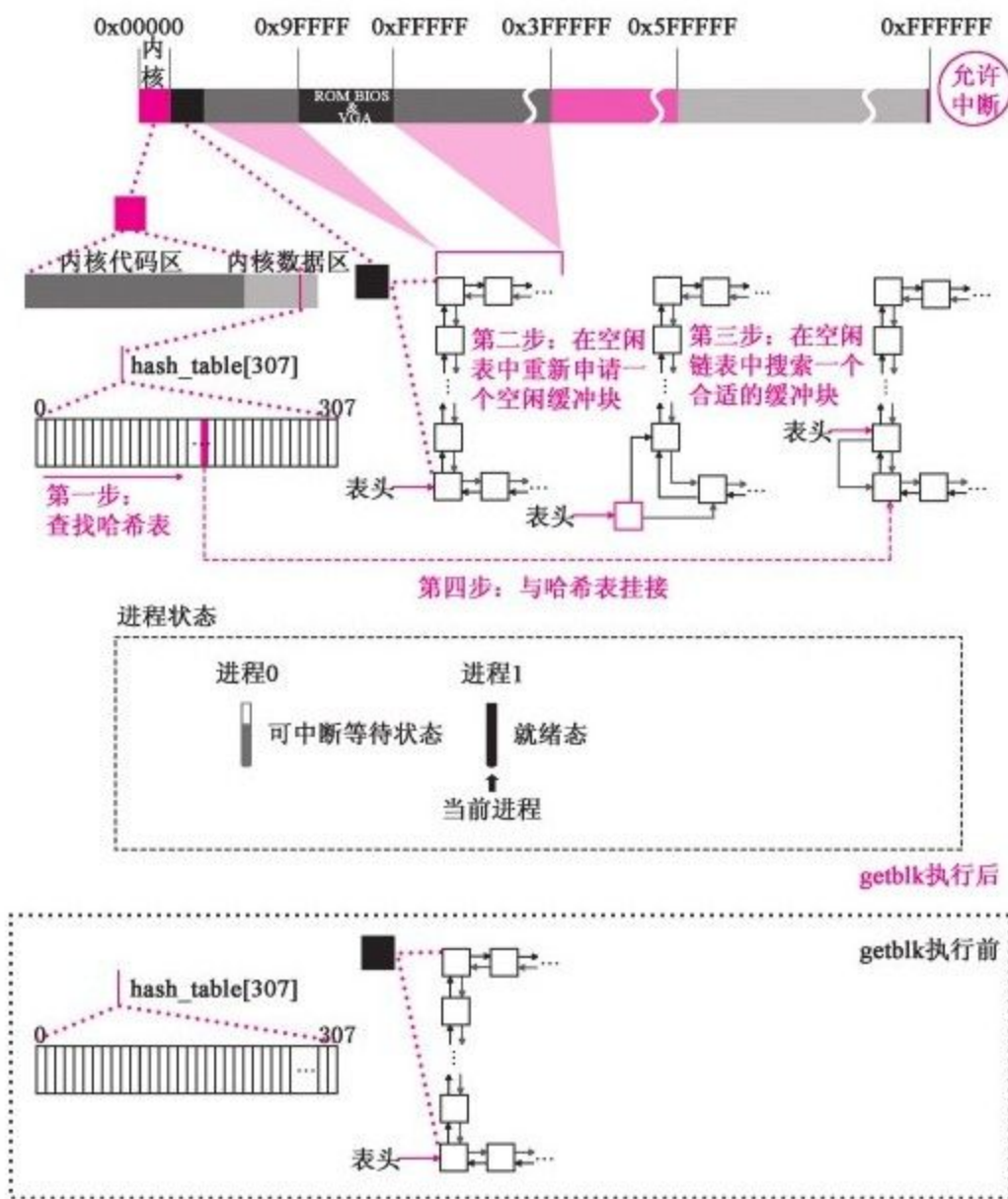


图 3-17 查找缓冲块

在`getblk ()`函数中，先调用`get_hash_table`
() 函数查找哈希表，检索此前是否有程序把现在要读的硬盘逻辑块（相同的设备号和块号）已经读到缓冲区。如果已经读到缓冲区，那就不用再费劲从硬盘上读取，直接用现成的，如图3-17中的第一步所示。使用哈希表进行查询的目的是提高查询速度。

执行代码如下：

```
//代码路径: fs/buffer.c:

.....

//在缓冲区得到与dev、block相符合或空闲的缓冲块。dev:
0x300、block: 0

struct buffer_head * getblk (int dev,int block)
```

```
{  
  
struct buffer_head * tmp, *bh;  
  
repeat:  
  
if (bh=get_hash_table (dev,block) )  
  
return bh;  
  
tmp=free_list;  
  
do{  
  
if (tmp->b_count)  
  
continue;  
  
if ( ! bh||BADNESS (tmp) <BADNESS (bh) ) {  
  
bh=tmp;  
  
if ( ! BADNESS (tmp) )  
  
break;  
  
}  
  
.....  
  
}
```

进入get_hash_table () 函数后，调用
find_buffer () 函数查找缓冲区中是否有指定设备
号、块号的缓冲块。如果能找到指定缓冲块，就
直接用。

执行代码如下：

```
//代码路径： fs/buffer.c:

.....

//查找哈希表，确定缓冲区中是否有指定dev、block的缓冲块。
dev: 0x300、block: 0

struct buffer_head * get_hash_table (int dev,int block)

{

struct buffer_head * bh;

for (; ) {

if (! (bh=fnd_buffer (dev,block) ) )

return NULL; //现在是第一次使用，肯定没有已经读到缓冲区的
块
```

```
    bh->b_count++;

    wait_on_buffer (bh) ;

    if (bh->b_dev==dev&&bh->b_blocknr==block)

        return bh;

    bh->b_count--;

}

}
```

现在是第一次使用缓冲区，缓冲区中不可能存在已读入的缓冲块，也就是说hash_table中没有挂接任何节点，find_buffer（）返回的一定是NULL。

执行代码如下：

```
//代码路径： fs/buffer.c:
```

```
.....
```


//NR_HASH是307, 对于dev: 0x300、block: 0而言, _hashfn (dev,block) 的值是154

```
#define _hashfn (dev,block) ( ( (unsigned) (dev^block) )  
%NR_HASH)
```

```
#define hash (dev,block) hash_table[_hashfn (dev,block) ]
```

.....

//在缓冲区查找指定dev、block的缓冲块

```
static struct buffer_head * fnd_buffer (int dev,int block)
```

```
{
```

```
    struct buffer_head * tmp;
```

```
    for (tmp=hash (dev,block) ; tmp !=NULL; tmp=tmp->  
b_next) //现在tmp->b_next为NULL
```

```
        if (tmp->b_dev==dev && tmp->b_blocknr==block)
```

```
            return tmp;
```

```
    return NULL;
```

```
}
```

从find_buffer () 、 get_hash_table () 函数退出后，返回getblk () 函数，在空闲表中申请一个新的空闲缓冲块。现在所有缓冲块都是绑定在空闲表中的，所以要在空闲表中申请新的缓冲块，如图3-17中的第二步所示。

执行代码如下：

//代码路径： fs/buffer.c:

```
#define BADNESS (bh) ( ( (bh) -> b_dirt << 1) + (bh) ->  
b_lock) //现在b_dirt、b_lock是0， BADNESS (bh) 就是00
```

```
struct buffer_head * getblk (int dev,int block)
```

```
{
```

```
struct buffer_head * tmp, *bh;
```

```
repeat:
```

```
if (bh=get_hash_table (dev,block) )
```

```
return bh;
```

```

tmp=free_list;

do{

if (tmp->b_count) //tmp->b_count现在为0

continue;

if (! bh||BADNESS (tmp) < BADNESS (bh) ) //bh现在为0

bh=tmp;

if (! BADNESS (tmp) ) //现在BADNESS (tmp) 是00, 取得空闲的缓冲块!

break;

}

/*and repeat until we find something good*/

}while ( (tmp=tmp->b_next_free) !=free_list) ;

if (! bh) {//现在不会出现没有获得空闲缓冲块的情况

sleep_on (&buffer_wait) ;

goto repeat;

}

wait_on_buffer (bh) ; //缓冲块没有加锁

```

```
if (bh->b_count) //现在还没有使用缓冲块

goto repeat;

while (bh->b_dirt) { //缓冲块的内容没有被修改

sync _dev (bh->b_dev) ;

wait _on_buffer (bh) ;

if (bh->b_count)

goto repeat;

}

if (find_buffer (dev,block) ) //现在虽然获得了空闲缓冲块，但并没有挂接到hash表中

goto repeat;

.....
```

申请到缓冲块后，对它进行初始化设置，并将这个空闲块挂接到hash_table上。执行代码如下：

//代码路径: fs/buffer.c:

```
struct buffer_head * getblk (int dev,int block)

{

.....

if (find_buffer (dev,block) )

goto repeat;

bh->b_count=1; //占用

bh->b_dirt=0;

bh->b_uptodate=0;

remove_from_queues (bh) ;

bh->b_dev=dev;

bh->b_blocknr=block;

insert_into_queues (bh) ;

return bh;

}
```

为了更容易理解代码，我们将这个过程分步图示出来，如图3-18所示。

//代码路径: fs/buffer.c:

.....

```
static inline void remove_from_queues (struct buffer_head * bh)
```

```
{
```

```
/*remove from hash-queue*/
```

```
if (bh->b_next) //bh->b_next现在为NULL
```

```
bh->b_next->b_prev=bh->b_prev;
```

```
if (bh->b_prev) //bh->b_prev现在为NULL
```

```
bh->b_prev->b_next=bh->b_next;
```

```
if (hash (bh->b_dev,bh->b_blocknr) ==bh) //现在不出现这个情况
```

```
hash (bh->b_dev,bh->b_blocknr) =bh->b_next;
```

```
/*remove from free list*/
```

```
if (! (bh->b_prev_free) ||! (bh->b_next_free) ) //正常时不会出现
```

```
panic ("Free block list corrupted") ;
```

```
bh->b_prev_free->b_next_free=bh->b_next_free;
```

```
bh->b_next_free->b_prev_free=bh->b_prev_free;
```

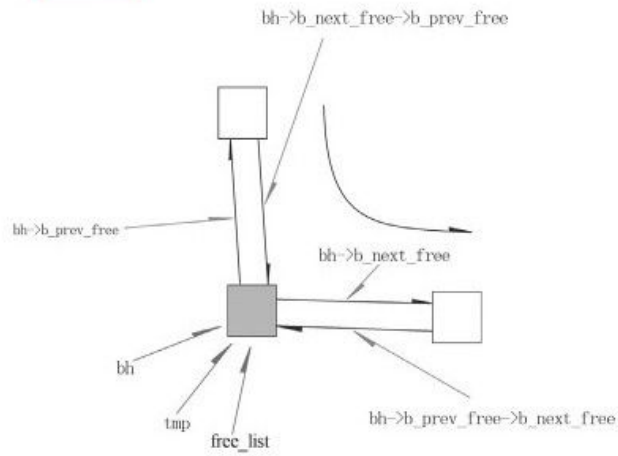
```
if (free_list==bh)
```

```
free_list=bh->b_next_free;
```

```
}
```

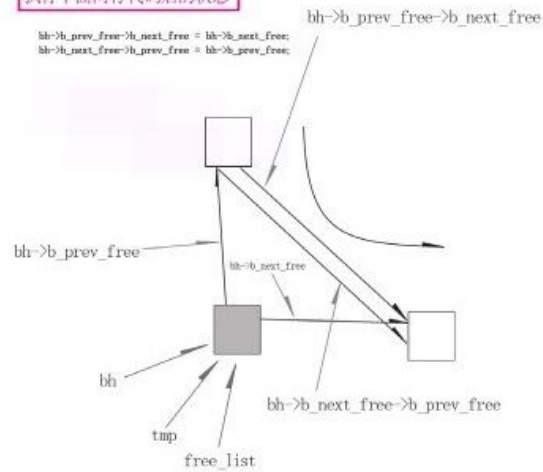
```
.....
```

初始状态



执行下面两行代码后的状态

```
bh->b_prev_free->b_next_free = bh->b_next_free;
bh->b_next_free->b_prev_free = bh->b_prev_free;
```



执行下面代码后的状态

```
free_list = bh->b_next_free
```

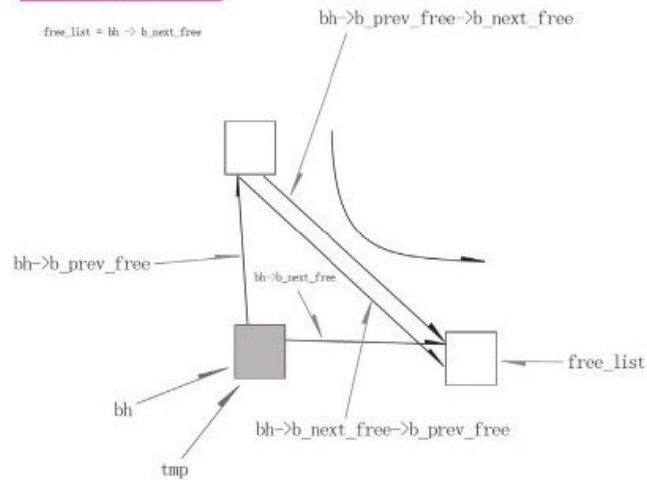


图 3-18 分步示意图

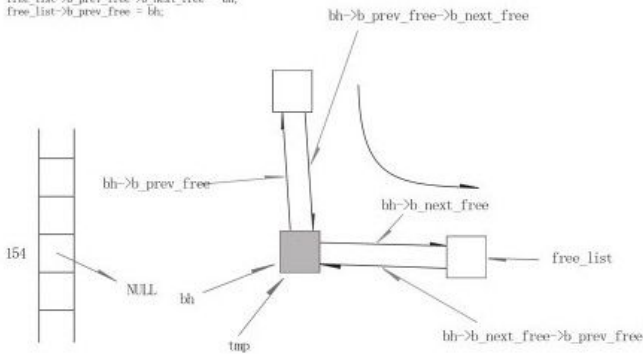
挂接hash_table的执行代码如下，分步示意图如图3-19所示。

执行下面代码后的状态

```

bh->b_next_free = free_list;
bh->b_prev_free = free_list->b_prev_free;
free_list->b_prev_free->b_next_free = bh;
free_list->b_prev_free = bh;

```

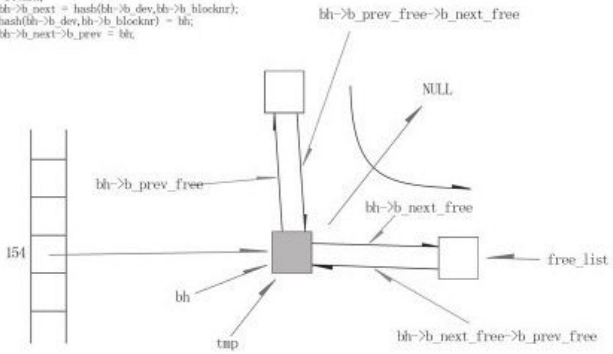


执行完下面代码后的状态

```

bh->b_prev = NULL;
bh->b_next = NULL;
if ((bh->b_dev)
    return;
bh->b_next = hash(bh->b_dev, bh->b_blocknr);
hash(bh->b_dev, bh->b_blocknr) = bh;
bh->b_next->b_prev = bh;

```



总效果图

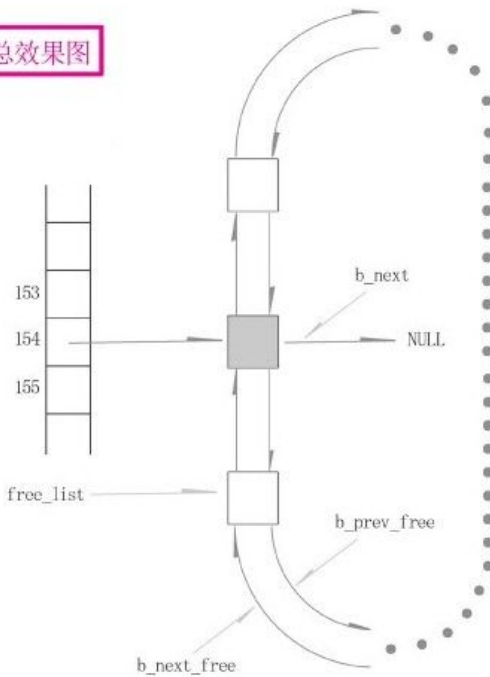


图 3-19 分布示意图

//代码路径: fs/buffer.c:

.....

static inline void insert_into_queues (struct buffer_head * bh)

{

/*put at end of free list*/

bh->b_next_free=free_list;

bh->b_prev_free=free_list->b_prev_free;

free_list->b_prev_free->b_next_free=bh;

free_list->b_prev_free=bh;

/*put the buffer in new hash-queue if it has a device*/

bh->b_prev=NULL;

bh->b_next=NULL;

if (! bh->b_dev)

return;

bh->b_next=hash (bh->b_dev,bh->b_blocknr) ;

```
hash (bh->b_dev,bh->b_blocknr) =bh;  
  
bh->b_next->b_prev=bh;  
  
}  
  
.....
```

执行完**getblk**（）函数后，返回**bread**（）函数。

3.将找到的缓冲块与请求项挂接

返回**bread**（）函数后，调用**ll_rw_block**（）这个函数，将缓冲块与请求项结构挂接，如图3-20所示。

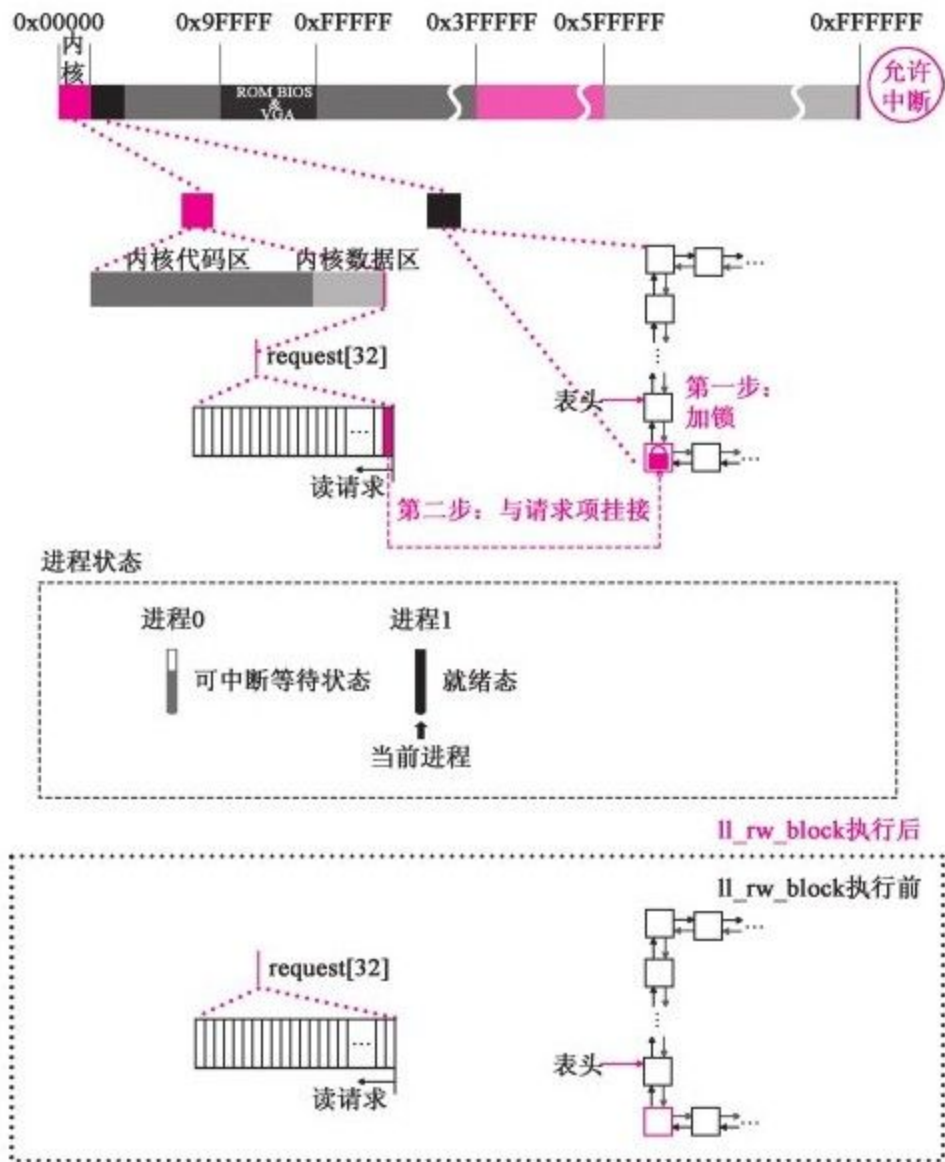


图 3-20 缓冲块与请求项挂接

执行代码如下：

//代码路径：fs/buffer.c:

```

struct buffer_head * bread (int dev,int block)

{

struct buffer_head * bh;

if ( ! (bh=getblk (dev,block) ) )

panic ("bread: getblk returned NULL\n") ;

if (bh->b_uptodate) //新申请的缓冲区肯定没有更新过

return bh;

ll_rw_block (READ,bh) ;

wait_on_buffer (bh) ;

if (bh->b_uptodate)

return bh;

brelse (bh) ;

return NULL;

}

```

进入ll_rw_block () 函数后，先判断缓冲块对应的设备是否存在或这个设备的请求项函数是

否挂接正常。如果存在且正常，说明可以操作这个缓冲块，调用make_request（）函数，准备将缓冲块与请求项建立关系，执行代码如下：

```
//代码路径: kernel/blk_dev/ll_rw_block.c:

void ll_rw_block (int rw,struct buffer_head * bh)

{

    unsigned int major;

    if ( (major=MAJOR (bh->b_dev) ) >
=NR_BLK_DEV||//NR_BLK_DEV是7，主设备号0-6，>=7意味着不存在

    ! (blk_dev[major].request_fn) ) {

        printk ("Trying to read nonexistent block-device\n\r") ;

        return;

    }

    make_request (major,rw,bh) ;

}
```

进程1继续执行，进入make_request（）函数后，先要将这个缓冲块加锁，目的是保护这个缓冲块在解锁之前将不再被任何进程操作，这是因为这个缓冲块现在已经被使用，如果此后再被挪作他用，里面的数据就会发生混乱。如图3-20右边的缓冲块buffer_head所示，其中选中的那个缓冲块对应的buffer_head已加锁。

之后，在请求项结构中，申请一个空闲请求项，准备与这个缓冲块相挂接。值得注意的是，如果是读请求，则从整个请求项结构的最末端开始寻找空闲请求项；如果是写请求，则从整个结构的2/3处，申请空闲请求项。这是因为从用户使用系统的心理角度讲，用户更希望读取的数据能更快地显现出来，所以给读取操作以更大的空

间。这时候，请求项结构是第一次被使用，而且是读请求，所以在请求项结构的末端找到一个空闲的请求项，如图3-20中的request[32]结构所示，其中的最后一项已被选中。之后，缓冲块与请求项正式挂接，并对这个请求项各个成员进行初始化。

执行代码如下：

```
//代码路径： kernel/blk_dev/ll_rw_block.c:

static inline void lock_buffer (struct buffer_head * bh)

{

cli () ;

while (bh->b_lock) //现在还没加锁

sleep_on (&bh->b_wait) ;

bh->b_lock=1; //加锁

sti () ;
```

```
}
```

```
.....
```

```
static void make_request (int major,int rw,struct buffer_head * bh) //
```

```
{
```

```
struct request * req;
```

```
int rw_ahead;
```

```
/*WRITEA/READA is special case-it is not really needed,so if the*/
```

```
/*buffer is locked,we just forget about it,else it's a normal read*/
```

```
if (rw_ahead= (rw==READA||rw==WRITEA) ) {
```

```
if (bh->b_lock) //现在还没有加锁
```

```
return;
```

```
if (rw==READA) //放弃预读写，改为普通读写
```

```
rw=READ;
```

```
else
```

```
rw=WRITE;
```

```
}
```

```
if (rw!=READ&&rw!=WRITE)
```

```

panic ("Bad block dev command,must be R/W/RA/WA") ;

lock_buffer (bh) ; //加锁

if ( (rw==WRITE&&! bh->b_dirt) || (rw==READ&&bh->
b_uptodate) ) {//现在还没有使用

unlock_buffer (bh) ;

return;

}

repeat:

/*we don't allow the write-requests to fill up the queue completely:

*we want some room for reads:  they take precedence.The last third

*of the requests are only for reads.

*/

if (rw==READ) //读从尾端开始, 写从2/3处开始

req=request+NR_REQUEST;

else

req=request+ ( (NR_REQUEST*2) /3) ;

/*find an empty request*/

```

```

    while (--req >= request) //从后向前搜索空闲请求项, 在
blk_dev_init中, dev初始化为-1, 即空闲

    if (req->dev < 0) //找到空闲请求项

        break;

    /*if none found,sleep on new requests:  check for rw_ahead*/

    if (req < request) {

        if (rw_ahead) {

            unlock_buffer (bh) ;

            return;

        }

        sleep_on (&wait_for_request) ;

        goto repeat;

    }

    /*fill up the request-info,and add it to the queue*/

    req->dev=bh->b_dev; //设置请求项

    req->cmd=rw;

    req->errors=0;

```

```
req->sector=bh->b_blocknr<<1;

req->nr_sectors=2;

req->buffer=bh->b_data;

req->waiting=NULL;

req->bh=bh;

req->next=NULL;

add_request (major+blk_dev,req) ;

}
```

调用add_request () 函数，向请求项队列中加载该请求项，进入add_request () 后，先对当前硬盘的工作情况进行分析，然后设置该请求项为当前请求项，并调用硬盘请求项处理函数

(dev->request_fn) () ，即do_hd_request () 函数去给硬盘发送读盘命令。图3-21中给出了请

求项管理结构与do_hd_request () 函数的对应关系。



图 3-21 将请求项与硬盘处理函数挂接

执行代码如下：

//代码路径： kernel/blk_dev/ll_rw_block.c:

```
static void add_request (struct blk_dev_struct * dev,struct request *  
req)
```

```

{

struct request * tmp;

req->next=NULL;

cli ( ) ;

if (req->bh)

req->bh->b_dirt=0;

if ( ! (tmp=dev->current_request) ) {

dev->current_request=req;

sti ( ) ;

(dev->request_fn) ( ) ; //do_hd_request ( )

return;

}

```

for (; tmp->next; tmp=tmp->next) //电梯算法的作用是让磁盘磁头的移动距离最小

```

if ( (IN_ORDER (tmp,req) ||

! IN_ORDER (tmp,tmp->next) ) &&

IN_ORDER (req,tmp->next) )

```

```
break;

req->next=tmp->next; //挂接请求项队列

tmp->next=req;

sti ();

}
```

4.读硬盘

进入do_hd_request () 函数去执行，为读盘做最后准备工作。具体的准备过程如图3-22所示。

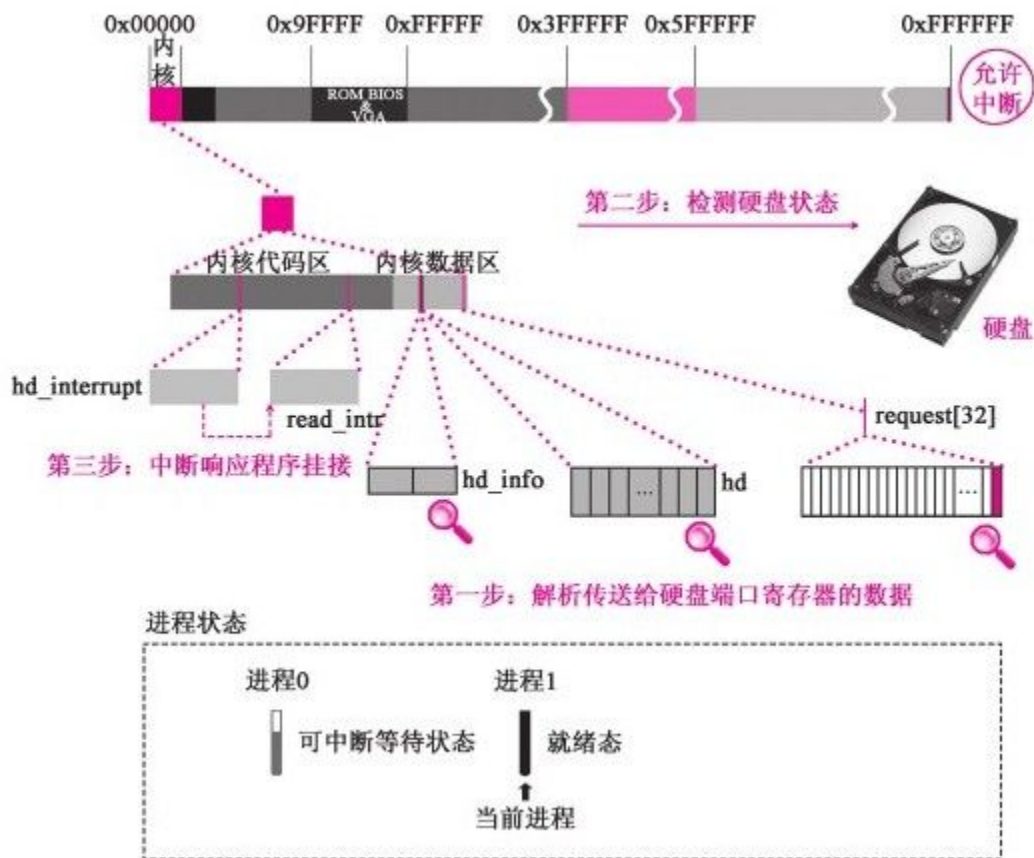


图 3-22 读盘操作前的主要准备工作

先通过对当前请求项数据成员的分析，解析出需要操作的磁头、扇区、柱面、操作多少个扇区……之后，建立硬盘读盘必要的参数，将磁头移动到0柱面，如图3-22中第二步所示；之后，针对命令的性质（读/写）给硬盘发送操作命令。现

在是读操作（读硬盘的引导块），所以接下来要调用hd_out（）函数来下达最后的硬盘操作指令。注意看最后两个实参，WIN_READ表示接下来要进行读操作，read_intr（）是读盘操作对应的中断服务程序，所以要提取它的函数地址，准备挂接，这一动作反映在图3-22中的第三步。请注意，这是通过hd_out（）函数实现的，读盘请求就挂接read_intr（）；如果是写盘，那就不是read_intr（），而是write_intr（）了。

执行代码如下：

```
//代码路径： kernel/blk_dev/hd.c:
```

```
void do_hd_request（void）
```

```
{
```

```
int i,r;
```

```

unsigned int block,dev;

unsigned int sec,head,cyl;

unsigned int nsect;

INIT _REQUEST;

dev=MINOR (CURRENT->dev) ;

block=CURRENT->sector;

if (dev >= 5*NR_HD || block+2 > hd[dev].nr_sects) {

end_request (0) ;

goto repeat;

}

block+=hd[dev].start_sect;

dev/=5;

__asm__
("divl%4": "=a" (block) , "=d" (sec) : "0" (block) , "1" (0) ,

"r" (hd_info[dev].sect) ) ;

__asm__
("divl%4": "=a" (cyl) , "=d" (head) : "0" (block) , "1" (0) ,

"r" (hd_info[dev].head) ) ;

```

```

sec++;

nsect=CURRENT->nr_sectors;

if (reset) {

reset=0; //置位，防止多次执行if (reset)

recalibrate=1; //置位，确保执行下面的if (recalibrate)

reset_hd (CURRENT_DEV) ; //将通过调用hd_out向硬盘发送
WIN_SPECIFY命令，建立硬盘读盘必要的参数

return;

}

if (recalibrate) {

recalibrate=0; //置位，防止多次执行if (recalibrate)

hd_out (dev,hd_info[CURRENT_DEV].sect, 0, 0, 0,

WIN_RESTORE, &recal_intr) ; //将向硬盘发送WIN_RESTORE
命令，将磁头移动到0柱面，以便从硬盘上读取数据

return;

}

if (CURRENT->cmd==WRITE) {

hd_out (dev,nsect,sec,head,cyl,WIN_WRITE, &write_intr) ;

```

```

    for (i=0; i<3000&&! (r=inb_p (HD_STATUS) &
DRQ_STAT) ; i++)

        /*nothing*/;

    if (! r) {

        bad_rw_intr () ;

        goto repeat;

    }

    port_write (HD_DATA,CURRENT->buffer, 256) ;

    }else if (CURRENT->cmd==READ) {

        hd_out (dev,nsect,sec,head,cyl,WIN_READ, &read_intr) ; //注意
        这两个参数

    }else

        panic ("unknown hd-command") ;

    }

```

进入hd_out () 函数中去执行读盘的最后一步：下达读盘指令，如图3-23中第一步所示。

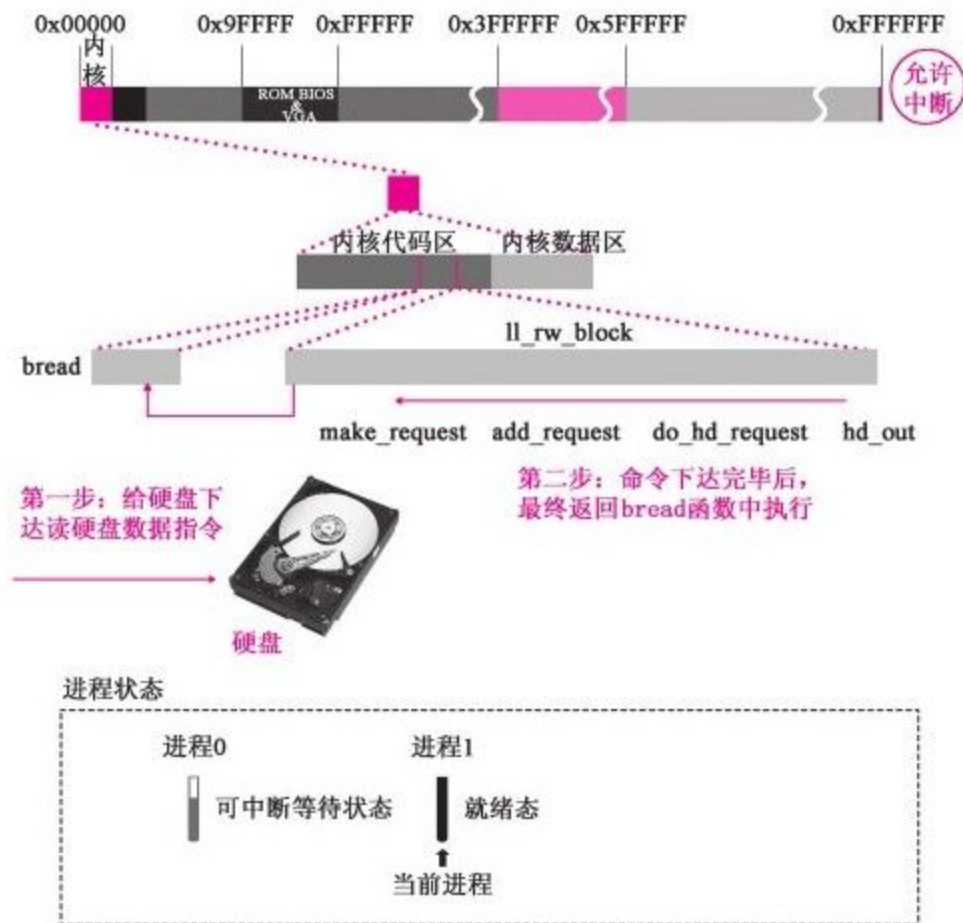


图 3-23 给硬盘端口寄存器传递参数

执行代码如下：

//代码路径：kernel/blk_dev/hd.c:

```
static void hd_out (unsigned int drive,unsigned int nsect,unsigned int sect,
```

```
unsigned int head,unsigned int cyl,unsigned int cmd,
```

```
void (*intr_addr) (void) ) //对比调用的传参WIN_READ, &
read_intr
```

```
{
```

```
register int port asm ("dx") ;
```

```
if (drive > 1 || head > 15)
```

```
panic ("Trying to write bad sector") ;
```

```
if (! controller_ready () )
```

```
panic ("HD controller not ready") ;
```

```
do_hd=intr_addr; //根据调用的实参决定是read_intr还是
write_intr, 现在是read_intr
```

```
outb _p (hd_info[drive].ctl, HD_CMD) ;
```

```
port=HD_DATA;
```

```
outb _p (hd_info[drive].wpcom >> 2, ++port) ;
```

```
outb _p (nsect, ++port) ;
```

```
outb _p (sect, ++port) ;
```

```
outb _p (cyl, ++port) ;
```

```
outb _p (cyl >> 8, ++port) ;
```

```
outb _p (0xA0 | (drive << 4) | head, ++port) ;
```

```
outb (cmd, ++port) ;

}

//代码路径: kernel/system_call.s:

_hd_interrupt:

.....

1: jmp 1f

1: xorl%edx, %edx

xchgl _do_hd, %edx

testl %edx, %edx

jne 1f

movl $_unexpected_hd_interrupt, %edx

.....
```

其中，do_hd=intr_addr；这一行是把读盘服务程序与硬盘中断操作程序相挂接，这里面的

do_hd是system_call.s中_hd_interrupt下面
xchgl_do_hd, %edx这一行所描述的内容。

现在要做读盘操作，所以挂接的就是实参
read_intr，如果是写盘，挂接的就应该是write_intr
() 函数。

下达读盘命令！

硬盘开始将引导块中的数据不断读入它的缓存中，同时，程序也返回了，将会沿着前面调用的反方向，即hd_out () 函数、do_hd_request
() 函数、add_request () 函数、make_request
() 函数、ll_rw_block () 函数，一直返回bread
() 函数中。

现在，硬盘正在继续读引导块。如果程序继续执行，则需要对引导块中的数据进行操作。但这些数据还没有从硬盘中读完，所以调用 `wait_on_buffer ()` 函数，挂起等待！

执行代码如下：

```
//代码路径: fs/buffer.c:
```

```
struct buffer_head * bread (int dev,int block)
```

```
{
```

```
struct buffer_head * bh;
```

```
if (! (bh=getblk (dev,block) ) )
```

```
panic ("bread: getblk returned NULL\n") ;
```

```
if (bh->b_uptodate)
```

```
return bh;
```

```
ll_rw_block (READ,bh) ;
```

```
wait_on_buffer (bh) ; //将等待缓冲块解锁的进程挂起
```

```
if (bh->b_uptodate)

return bh;

brelse (bh) ;

return NULL;

}
```

进入wait_on_buffer () 函数后，判断刚才申请到的缓冲块是否被加锁。现在，缓冲块确实加锁了，调用sleep_on () 函数。如图3-24中的第二步所示。执行代码如下：

//代码路径： fs/buffer.c:

```
static inline void wait_on_buffer (struct buffer_head * bh)

{

cli () ;

while (bh->b_lock) //前面已经加锁

sleep _on (&bh->b_wait) ;
```

```
sti ();  
  
}
```

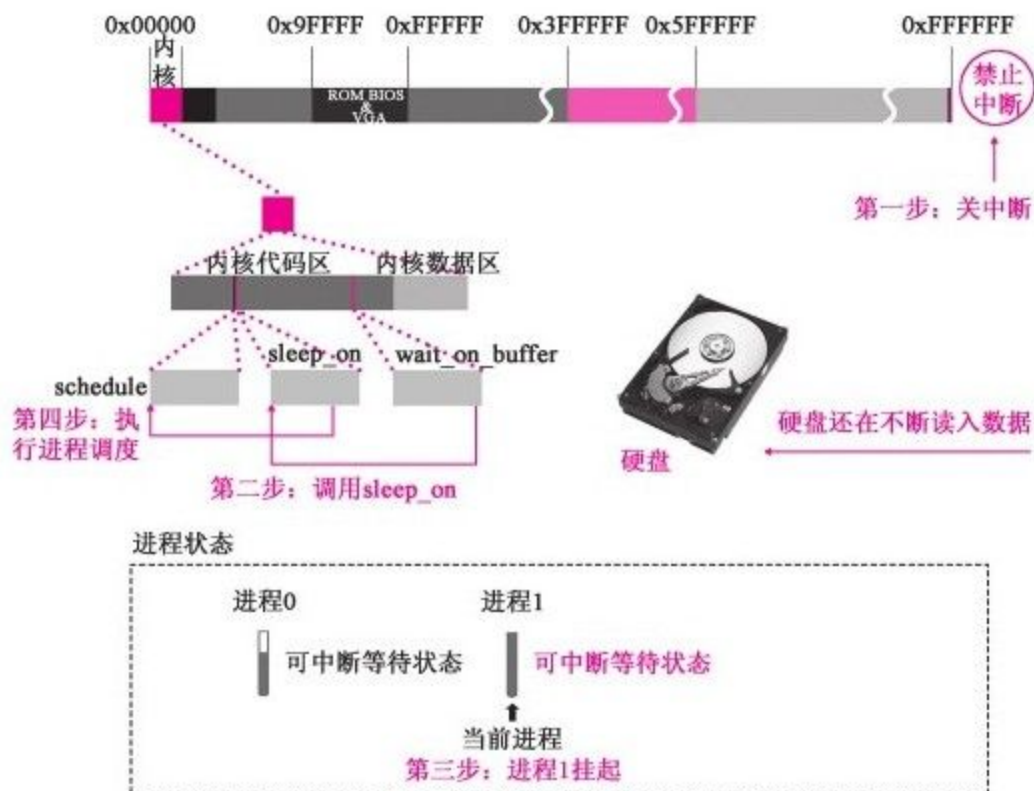


图 3-24 进程1挂起并执行调度

进入sleep_on () 函数后，将进程1设置为不可中断等待状态，如图3-24中第三步所示，进程1

挂起；然后调用schedule（）函数，准备进程切换，执行代码如下：

```
//代码路径： kernel/sched.c:

void sleep_on (struct task_struct ** p)

{

    struct task_struct * tmp;

    if (! p)

        return;

    if (current==& (init_task.task) )

        panic ("task[0]trying to sleep") ;

    tmp=*p;

    *p=current;

    current->state=TASK_UNINTERRUPTIBLE;

    schedule () ;

    if (tmp)

        tmp->state=0;
```

进入schedule（）函数后，切换到进程0去执行。图3-25给出了切换过程的主要步骤。

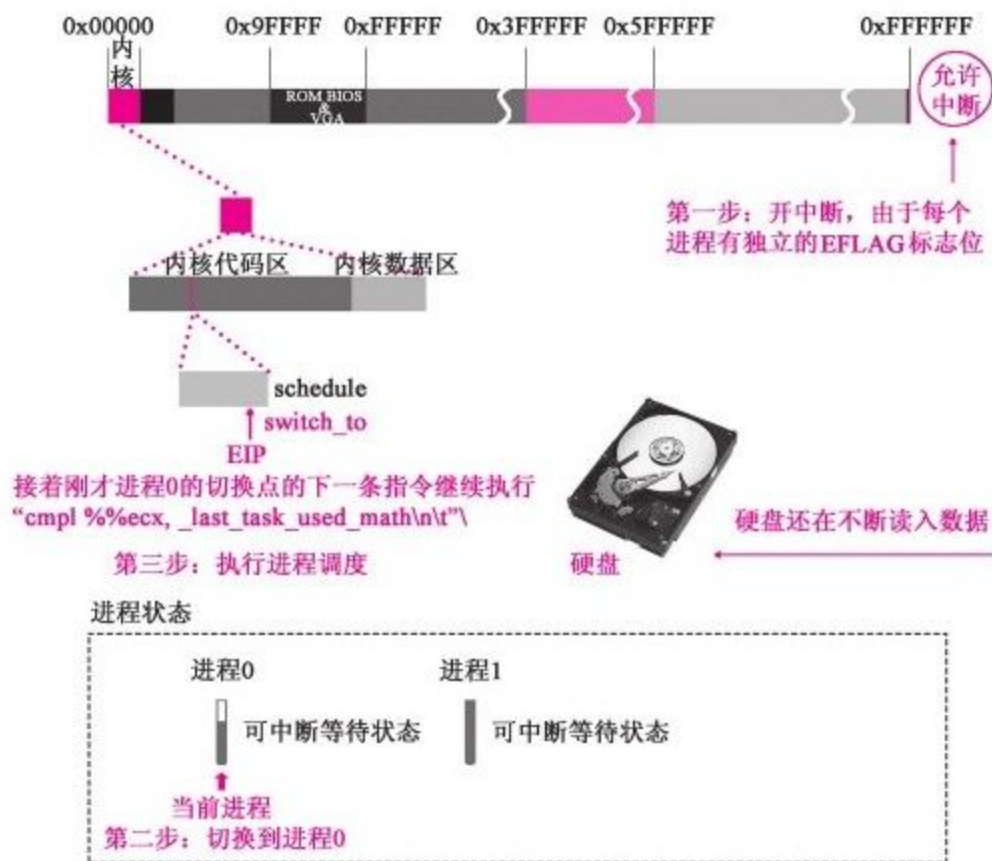


图 3-25 切换到进程0去执行

具体执行步骤在3.2节中已经说明。但第二次遍历task[64]的时候，与3.2节中执行的结果不一样。此时只有两个进程，进程0的状态是可中断等待状态，进程1的状态也已经刚刚被设置成了不可中断等待状态。常规的进程切换条件是，剩余时间片最多且必须是就绪态，即代码“if ((*p) -> state==TASK_RUNNING && (*p) -> counter > c) ”给出的条件。现在两个进程都不是就绪态，按照常规的条件无法切换进程，没有进程可以执行。

这是一个非常尴尬的状态。

操作系统的设计者对这种状态的解决方案是：强行切换到进程0！

注意：c的值将仍然是-1，所以next仍然是0，这个next就是要切换到进程的进程号。可以看出，如果没有合适的进程，next的数值将永远是0，就会切换到进程0去执行！

执行代码如下：

```
//代码路径： kernel/sched.c:
```

```
void schedule (void)
```

```
{
```

```
.....
```

```
while (1) {
```

```
c=-1;
```

```
cnext=0;
```



```

ci=NR_TASKS;

cp=&task[NR_TASKS];

cwhile (--i) {

cif (! *--p)

continue;

if ( (*p) -> state==TASK_RUNNING&& (*p) -> counter>c)

c= (*p) -> counter,next=i;

}

if (c) break;

for (p=&LAST_TASK; p>&FIRST_TASK; --p)

if (*p)

    (*p) -> counter= ( (*p) -> counter>>1) +

    (*p) -> priority;

}

switch_to (next) ; //next是0!

}

```

调用switch_to (0) 执行代码如下:

```
//代码路径: kernel/sched.h:
```

```
#define switch_to (n) {\
```

```
struct{long a,b; }__tmp; \
```

```
__asm__ ("cmpl%%ecx, _current\n\t"\
```

```
"je 1f\n\t"\
```

```
"movw%%dx, %1\n\t"\
```

```
"xchgl%%ecx, _current\n\t"\
```

"ljmp%0\n\t"\\跳转到进程0, 参看3.2节中有关switch_to (n) 的讲解及代码解释

```
"cmpl%%ecx, _last_task_used_math\n\t"\
```

```
"jne 1f\n\t"\
```

```
"clts\n\t"\
```

```
"1: "\
```

```
: "m" (*&__tmp.a) , "m" (*&__tmp.b) , \
```

```
"d" (_TSS (n) ) , "c" ( (long) task[n] ) ) ; \
```

}

`switch_to (0)` 执行完后，已经切换到进程0去执行。前面3.2节中已经说明，当时进程0切换到进程1时，是从`switch_to (1)` 的"`ljmp%0\n\t`"这一行切换走的，TSS中保存当时的CPU所有寄存器的值，其中CS、EIP指向的就是它的下一行，所以，现在进程0要从"`cmpl%%ecx, _last_task_used_math\n\t`"这行代码开始执行，如图3-25中的第三步所示。

将要执行的代码如下：

```
//代码路径: kernel/sched.h:

#define switch_to (n) {\

    struct{long a,b; }__tmp;

    __asm__ ("cmpl%%ecx, _current\n\t"
```

```
"je 1f\n\t\"
```

```
"movw%%dx, %1\n\t\"
```

```
"xchgl%%ecx, _current\n\t\"
```

```
"ljmp%0\n\t\"
```

"cmpl%%ecx, _last_task_used_math\n\t"\\从这一行开始执行，此时是进程0在执行，0特权级

```
"jne 1f\n\t\"
```

```
"clts\n\"
```

```
"1: \"
```

```
: "m" (*&__tmp.a) , "m" (*&__tmp.b) , \"
```

```
"d" (_TSS (n) ) , "c" ( (long) task[n] ) ) ; \"
```

```
}
```

回顾3.2节，当时进程0切换到进程1是从pause
() 、 sys_pause () 、 schedule () 、 switch_to
(1) 这个调用路线执行过来的。现在， switch_to

(1) 后半部分执行完毕后，就应该返回`sys_pause`
`()`、`for (;) pause ()`中执行了。

`pause`这个函数将在`for (;)`这个循环里面被反复调用，所以，会继续调用`schedule`函数进行进程切换。而再次切换的时候，由于两个进程还都不是就绪态，按照前面讲述过的理由，当所有进程都挂起的时候，内核会执行`switch_to`强行切换到进程0。

现在，`switch_to`中情况有些变化，"`cmpl%%ecx, _current\n\t`" "`je 1f\n\t`"的意思是：如果切换到的进程就是当前进程，就跳转到下面的“1:”处直接返回。此时当前进程正是进程0，要切换到的进程也是进程0，正好符合这个条件。

执行代码如下：

```
//代码路径: init/main.c:
```

```
void main (void)
```

```
{
```

```
.....
```

```
for (; ) pause ();
```

```
}
```

```
//代码路径: kernel/sched.h:
```

```
#define switch_to (n) {\
```

```
struct{long a,b; }__tmp; \
```

```
__asm__ ("cmpl%%ecx, _current\n\t"\
```

```
"je 1f\n\t"\
```

```
"movw%%dx, %1\n\t"\
```

```
"xchgl%%ecx, _current\n\t"\
```

```
"ljmp%0\n\t"\
```

```
"cmpl%%ecx, _last_task_used_math\n\t"\
```

```
"jne 1f\n\t\  
  
"clts\n"  
  
"1:  \  
  
: "m" (*&__tmp.a) , "m" (*&__tmp.b) , \  
  
"d" (_TSS (n) ) , "c" ( (long) task[n] ) ) ; \  
  
}
```

所以，又回到进程0（注意：不是切换到进程0）。

循环执行这个动作，如图3-26所示。

从这里可以看出操作系统的设计者为进程0设计的特殊职能：当所有进程都挂起或没有任何进程执行的时候，进程0就会出来维持操作系统的基本运转，等待挂起的进程具备可执行的条件。业内人士也称进程0为怠速进程，很像维持汽车等待

驾驶员踩油门的怠速状态那样维护计算机的怠速状态。

注意：硬盘的读写速度远低于CPU执行指令的速度（2~3个量级）。现在，硬盘仍在忙着把指定的数据读到它的缓存中.....

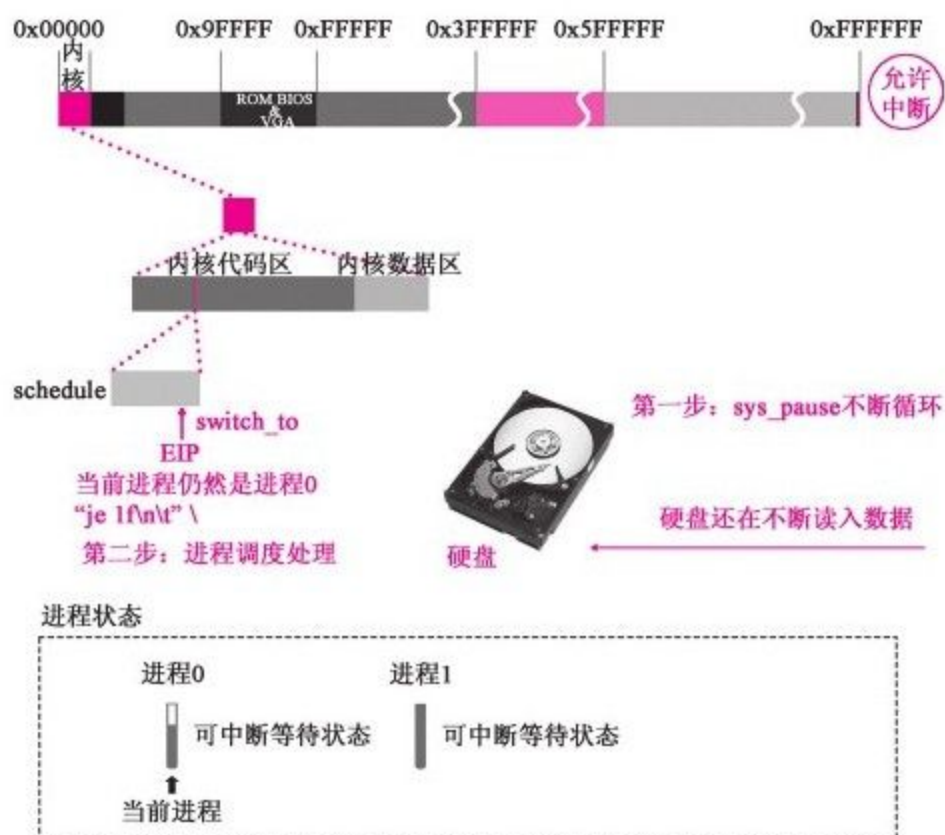


图 3-26 进程0的循环执行过程

6.进程0执行过程中发生硬盘中断

循环执行了一段时间后，硬盘在某一时刻把一个扇区的数据读出来了，产生硬盘中断。CPU 接到中断指令后，终止正在执行的程序，终止的位置肯定是在 `pause ()` 、 `sys_pause ()` 、 `schedule ()` 、 `switch_to (n)` 循环里面的某行指令处，如图3-27中的第一步所示。

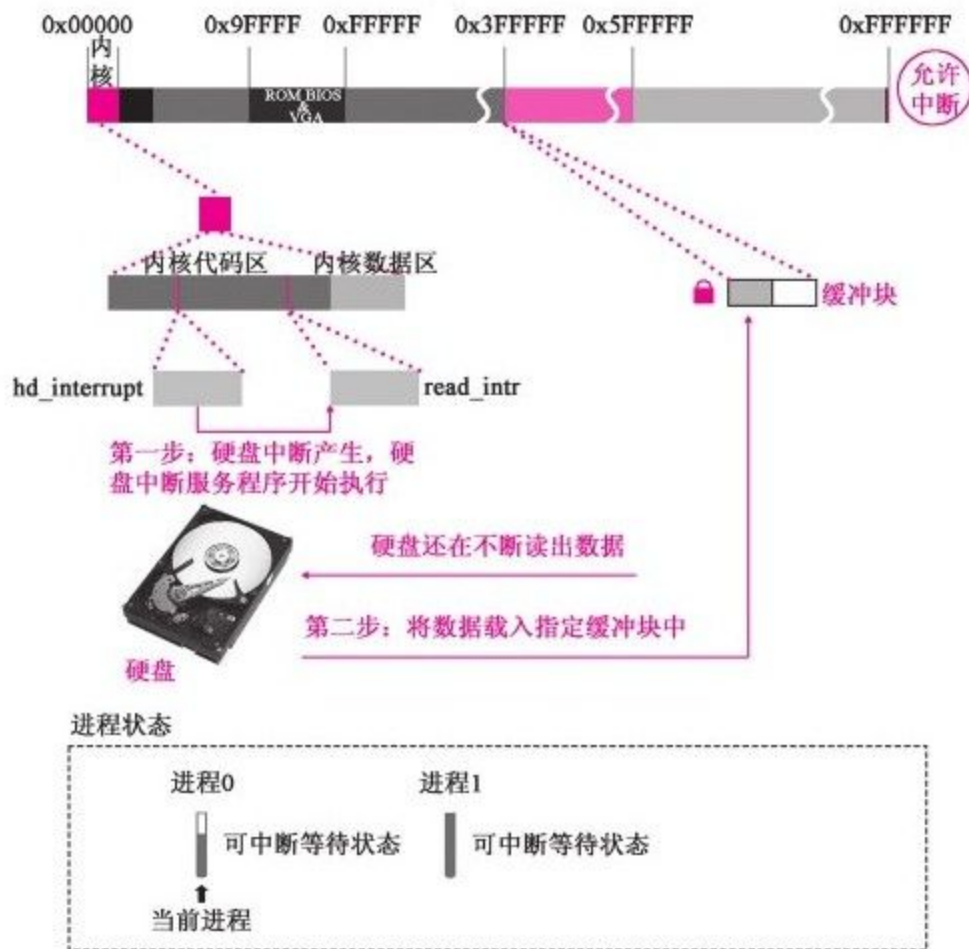


图 3-27 硬盘中断处理过程

然后转去执行硬盘中断服务程序。执行代码如下：

```
//代码路径： kernel/system_call.s:
```

.....

_hd_interrupt:

pushl %eax//保存CPU的状态

pushl %ecx

pushl %edx

push %ds

push %es

push %fs

movl \$0x10, %eax

mov %ax, %ds

mov %ax, %es

movl \$0x17, %eax

mov %ax, %fs

movb \$0x20, %al

outb %al, \$0xA0

jmp 1f

1: jmp 1f

1: xorl%edx, %edx

```
xchgl _do_hd, %edx

testl %edx, %edx

jne 1f

movl $_unexpected_hd_interrupt, %edx

1: outb%al, $0x20

call*%edx

.....
```

别忘了中断会自动压栈ss、esp、eflags、cs、eip，硬盘中断服务程序的代码接着将一些寄存器的数据压栈以保存程序的中断处的现场。之后，执行_do_hd处的读盘中断处理程序，对应的代码应该是call*%edx这一行。这个edx里面是读盘中断处理程序read_intr的地址，参看hd_out（）函数的讲解及代码注释。

`read_intr` () 函数会将已经读到硬盘缓存中的数据复制到刚才被锁定的那个缓冲块中（注意：锁定是阻止进程方面的操作，而不是阻止外设方面的操作），这时1个扇区256字（512字节）的数据读入前面申请到的缓冲块，如图3-27中的第二步所示。执行代码如下：

//代码路径：kernel/blk_dev/hd.c:

```
static void read_intr (void)
```

```
{
```

```
if (win_result ( ) ) {
```

```
bad_rw_intr ( ) ;
```

```
do_hd_request ( ) ;
```

```
return;
```

```
}
```

```
port_read (HD_DATA,CURRENT->buffer, 256) ;
```

```
CURRENT->errors=0;

CURRENT->buffer+=512;

CURRENT->sector++;

if (--CURRENT->nr_sectors) {

    do_hd=&read_intr;

    return;

}

end_request (1) ;

do_hd_request () ;

}
```

但是，引导块的数据是1024字节，请求项要求的也是1024字节，现在仅读出了一半，硬盘会继续读盘。与此同时，在得知请求项对应的缓冲块数据没有读完的情况下，内核将再次把

`read_intr ()` 绑定在硬盘中断服务程序上，以待下次使用，之后中断服务程序返回。

进程1仍处在被挂起状态，`pause ()` 、
`sys_pause ()` 、`schedule ()` 、`switch_to (0)` 循环从刚才硬盘中断打断的地方继续循环，硬盘继续读盘.....

整个过程如图3-28所示。



图 3-28 进程0继续循环执行

又过了一段时间后，硬盘剩下的那一半数据也读完了，硬盘产生中断，读盘中断服务程序再次响应这个中断，进入`read_intr()`函数后，仍然会判断请求项对应的缓冲块的数据是否读完了，对应代码如下：

//代码路径: kernel/blk_dev/hd.c:

```
static void read_intr (void)

{

.....

if (--CURRENT->nr_sectors)

.....

end_request (1) ;

.....

}
```

这次已经将请求项要求的数据量全部读完了，经检验确认完成后，不执行if里面的内容了，跳到end_request () 函数去执行，如图3-29中read_intr () 这个函数所示。

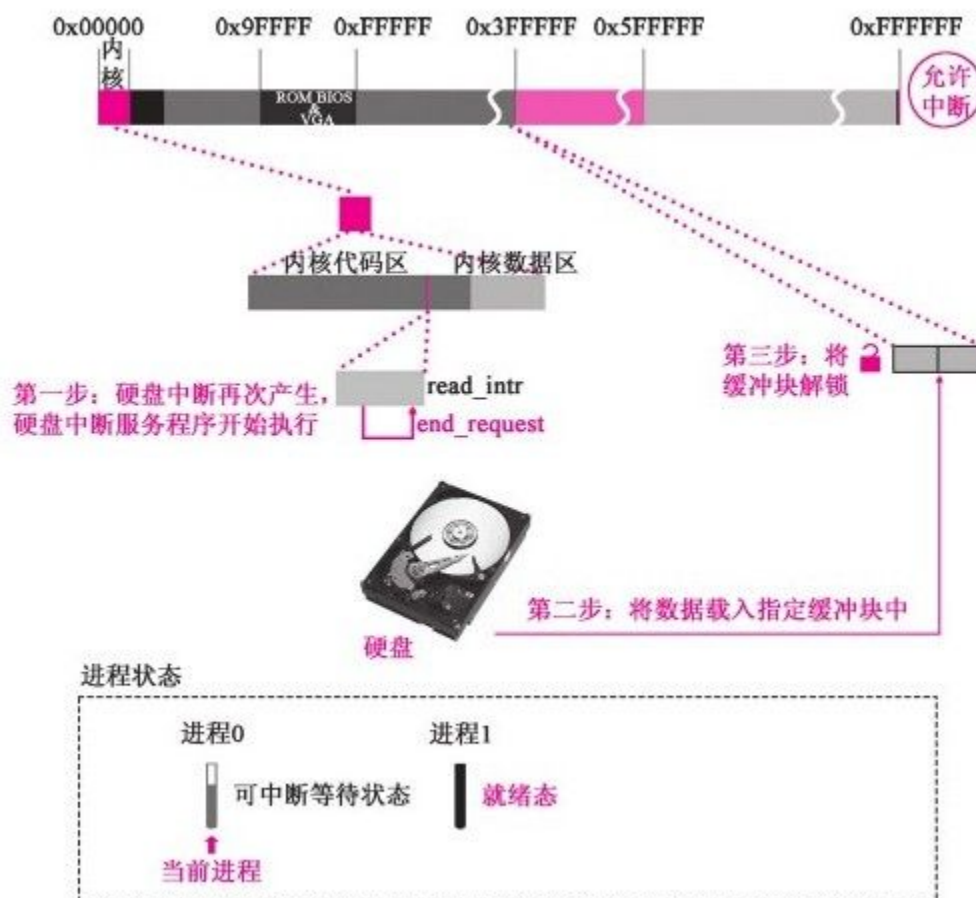


图 3-29 再次响应硬盘中断并唤醒进程1

进入`end_request()`后, 由于此时缓冲块的内容已经全部读进来了, 将这个缓冲块的更新标志`b_uptodate`置1, 说明它可用了, 执行代码如下:

//代码路径: kernel/blk_dev/blk.h:

```
extern inline void end_request (int uptodate)

{

DEVICE _OFF (CURRENT->dev) ;

if (CURRENT->bh) {

CURRENT->bh->b_uptodate=uptodate; //uptodate是参数, 为1

unlock_buffer (CURRENT->bh) ;

}

if (! uptodate) {

printk (DEVICE_NAME"I/O error\n\r") ;

printk ("dev%04x,block%d\n\r", CURRENT->dev,

CURRENT->bh->b_blocknr) ;

}

wake_up (&CURRENT->waiting) ;

wake_up (&wait_for_request) ;

CURRENT->dev=-1;

CURRENT=CURRENT->next;
```

```
}
```

之后，调用unlock_buffer（）函数为缓冲块解锁。在unlock_buffer（）函数中调用wake_up（）函数，将等待这个缓冲块解锁的进程（进程1）唤醒（设置为就绪态），并对刚刚使用过的请求项进行处理，如将它对应的请求项设置为空闲.....执行代码如下：

```
//代码路径： kernel/blk_dev/blk.h:
```

```
extern inline void unlock_buffer (struct buffer_head * bh)

{

if (! bh->b_lock)

printk (DEVICE_NAME": free buffer being unlocked\n") ;

bh->b_lock=0;

wake_up (&bh->b_wait) ;

}
```

//代码路径: kernel/sched.c:

```
void wake_up (struct task_struct ** p)

{

if (p && *p) {

    (** p) .state=0; //设置为就绪态

    *p=NULL;

}

}
```

硬盘中断处理结束，也就是载入硬盘引导块的工作结束后，计算机在`pause ()`、`sys_pause ()`、`schedule ()`、`switch_to (0)`循环中继续执行，如图3-29中第三步所示。

7.读盘操作完成后，进程调度切换到进程1执行

现在，引导块的两个扇区已经载入内核的缓冲块，进程1已经处于就绪态。注意：虽然进程0一直参与循环运行，但它是非就绪态。现在只有进程0和进程1，当循环执行到schedule函数时就会切换进程1去执行。该过程如图3-30所示。

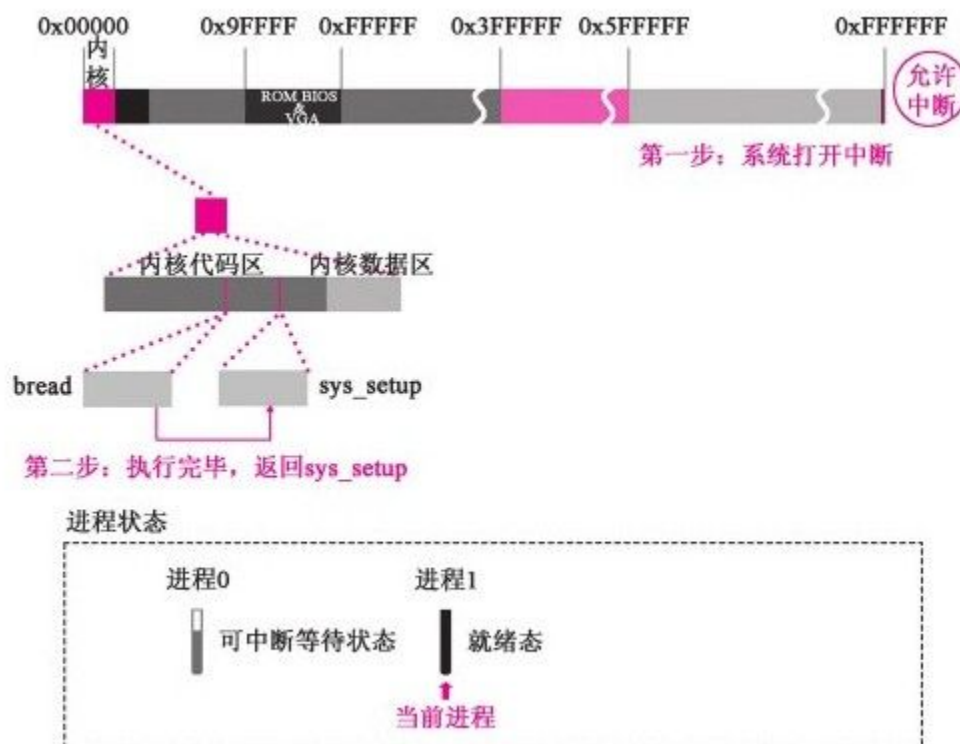


图 3-30 切换到进程1，并返回sys_setup

切换到进程1后，进程1从下面的代码继续执行：

//代码路径： kernel/sched.h:

```
#define switch_to (n) {\
```

```
struct{long a,b; }__tmp; \
```

```
__asm__ ("cmpl%%ecx, _current\n\t"\
```

```
"je 1f\n\t"\
```

```
"movw%%dx, %1\n\t"\
```

```
"xchgl%%ecx, _current\n\t"\
```

```
"ljmp%0\n\t"\
```

```
"cmpl%%ecx, _last_task_used_math\n\t"//理由和前面讲述的  
switch_to一样
```

```
"jne 1f\n\t"\
```

```
"clts\n\t"\
```

```
"1: "\
```

```
: "m" (*&__tmp.a) , "m" (*&__tmp.b) , \
```

```
"d" ( _TSS (n) ) , "c" ( (long) task[n] ) ) ; \
}
```

可以看出，所有进程间的切换都是这个模式。

进程1是从"ljmp%0\n\t"切换走的，所以现在执行它的下一行。现在，返回切换的发起者sleep_on（）函数中，并最终返回bread（）函数中。在bread（）函数中判断缓冲块的b_uptodate标志已被设置为1，直接返回，bread（）函数执行完毕。执行代码如下：

```
//代码路径： fs/buffer.c:
```

```
struct buffer_head * bread (int dev,int block)
{
    struct buffer_head * bh;
```



```
if ( ! (bh=getblk (dev,block) ) )  
  
panic ("bread: getblk returned NULL\n") ;  
  
if (bh->b_uptodate)  
  
return bh;  
  
ll_rw_block (READ,bh) ;  
  
wait_on_buffer (bh) ;  
  
if (bh->b_uptodate)  
  
return bh;  
  
brelse (bh) ;  
  
return NULL;  
  
}
```

回到sys_setup函数继续执行，处理硬盘引导块载入缓冲区后的事务。缓冲块里面装载着硬盘的引导块的内容，先来判断硬盘信息有效标志'55AA'。如果第一个扇区的最后2字节不

是'55AA'，就说明这个扇区中的数据是无效的
（我们假设引导块的数据没有问题）。执行代码
如下：

```
//代码路径: kernel/blk_dev/hd.c:

int sys_setup (void * BIOS)

{

.....

for (drive=0; drive<NR_HD; drive++) {

if (! (bh=bread (0x300+drive*5, 0) ) ) {

printk ("Unable to read partition table of drive%d\n\r",

drive) ;

panic ("") ;

}

if (bh->b_data[510] != 0x55 || (unsigned char) //我们假设引导块
的数据没问题

bh->b_data[511] != 0xAA) {
```

```

    printk ("Bad partition table on drive%d\n\r", drive) ;

    panic ("") ;

}

p=0x1BE+ (void *) bh->b_data; //根据引导块中的分区信息设置
hd[]

for (i=1; i<5; i++, p++) {

    hd[i+5*drive].start_sect=p->start_sect;

    hd[i+5*drive].nr_sects=p->nr_sects;

}

brelse (bh) ; //释放缓冲块 (引用计数减1)

}

if (NR_HD)

    printk ("Partition table%s ok.\n\r", (NR_HD>1) ?"s": "") ;

.....

}

```

之后，利用从引导块中采集到的分区表信息来设置hd[]，如图3-31所示。

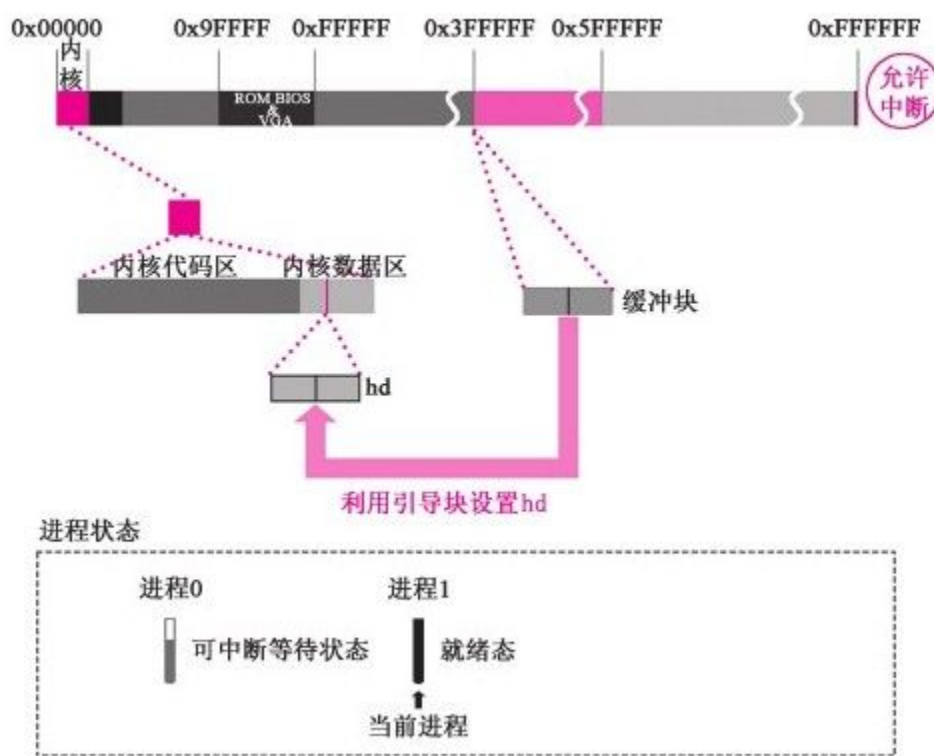


图 3-31 利用引导块设置硬盘分区管理结构

读引导块的缓冲块已经完成使命，调用**brelse**
() 函数释放，以便以后继续程序使用。

根据硬盘分区信息设置hd[], 为第5章安装硬盘文件系统做准备的工作都已完成。下面, 我们将介绍进程1用虚拟盘替代软盘使之成为根设备, 为加载根文件系统做准备。

3.3.2 进程1格式化虚拟盘并更换根设备为虚拟盘

第2章的2.3节设置了虚拟盘空间并初始化。那时的虚拟盘只是一块“白盘”，尚未经过类似“格式化”的处理，还不能当做一个块设备使用。格式化所用的信息就在boot操作系统的软盘上。第1章讲解过，第一个扇区是bootsect，后面4个扇区是setup，接下来的240个扇区是包含head的system模块，一共有245个扇区。“格式化”虚拟盘的信息从256扇区开始。

下面，进程1调用rd_load（）函数，用软盘上256以后扇区中的信息“格式化”虚拟盘，使之成为一个块设备。

执行代码如下：

```
//代码路径: kernel/blk_dev/hd.c:

int sys_setup (void * BIOS)

{

.....

if (NR_HD)

printk ("Partition table%s ok.\n\r", (NR_HD > 1) ? "s": "") ;

rd_load () ;

mount_root () ;

return (0) ;

}
```

进入rd_load () 函数后，调用breada () 函数从软盘预读一些数据块，也就是“格式化”虚拟盘需要的引导块、超级块。

注意：现在根设备是软盘。

`breada()` 和 `bread()` 函数类似，不同点在于可以把一些连续的数据块都读进来，一共三块，分别是257、256和258，其中引导块在256（尽管引导块并未实际使用）、超级块在257中。从软盘上读取数据块与`bread`读硬盘上的数据块原理基本一致，具体情况参看3.3.1节的讲解。读取完成后的状态如图3-32所示。可以看出3个连续的数据块被读入了高速缓冲区的缓冲块中，其中，超级块用红色框标注。

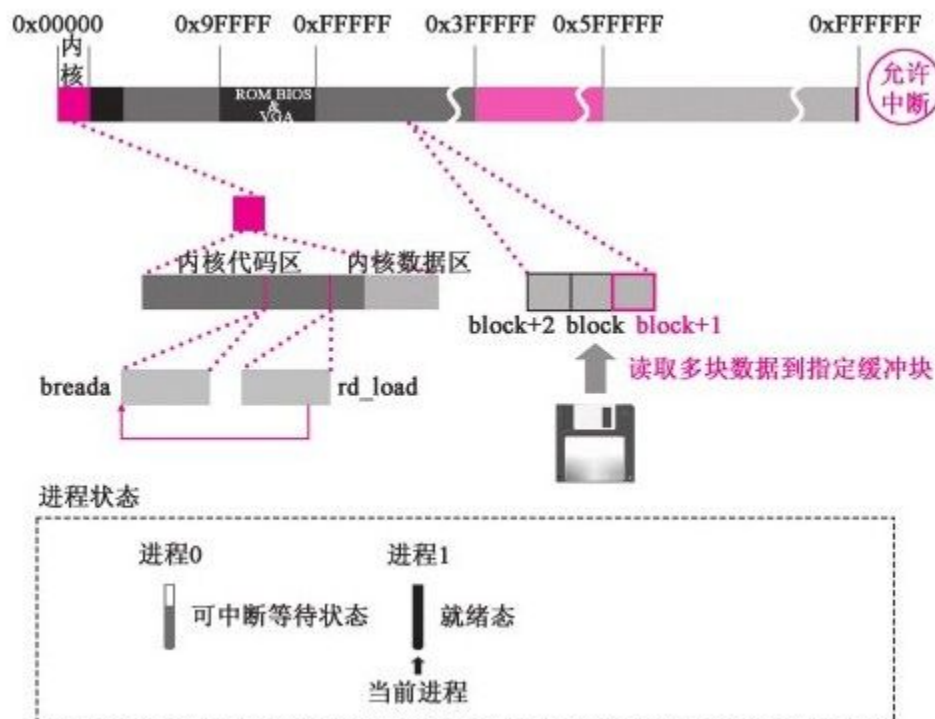


图 3-32 读取根文件系统超级块

之后，分析超级块信息，包括判断文件系统是不是minix文件系统、接下来要载入的根文件系统的数据块数会不会比整个虚拟盘区都大.....这些条件都通过，才能继续加载根文件系统。分析完毕，释放缓冲块。

整个过程如图3-33所示。

执行代码如下：

```
//代码路径: kernel/blk_dev/ramdisk.c:
```

```
void rd_load (void)
```

```
{
```

```
struct buffer_head * bh;
```

```
struct super_block s;
```

```
int block=256; /*Start at block 256*/
```

```
int I=1;
```

```
int nblocks;
```

```
char * cp; /*Move pointer*/
```

```
if (! rd_length)
```

```
return;
```

```
printk ("Ram disk:  %d bytes,starting at 0x%x\n", rd_length,
```

```
      (int) rd_start) ;
```

```
if (MAJOR (ROOT_DEV) !=2) //如果根设备不是软盘
```

```
return;
```

```

bh=breada (ROOT_DEV,block+1, block,block+2, -1) ;

if (! bh) {

printk ("Disk error while looking for ramdisk! \n") ;

return;

}

* ((struct d_super_block *) &s) =* ((struct d_super_block *)
bh->b_data) ;

brelse (bh) ;

if (s.s_magic !=SUPER_MAGIC) //如果不等, 说明不是minix文
件系统

/*No ram disk image present,assume normal floppy boot*/

return;

nblocks=s.s_nzones<<s.s_log_zone_size; //算出虚拟盘的块数

if (nblocks> (rd_length>>BLOCK_SIZE_BITS) ) {

printk ("Ram disk image too big! (%d blocks, %d avail) \n",

nblocks,rd_length>>BLOCK_SIZE_BITS) ;

return;

}

```

```

printk ("Loading%d bytes into ram disk.....0000k",
nblocks<<BLOCK_SIZE_BITS) ;

.....

}

```

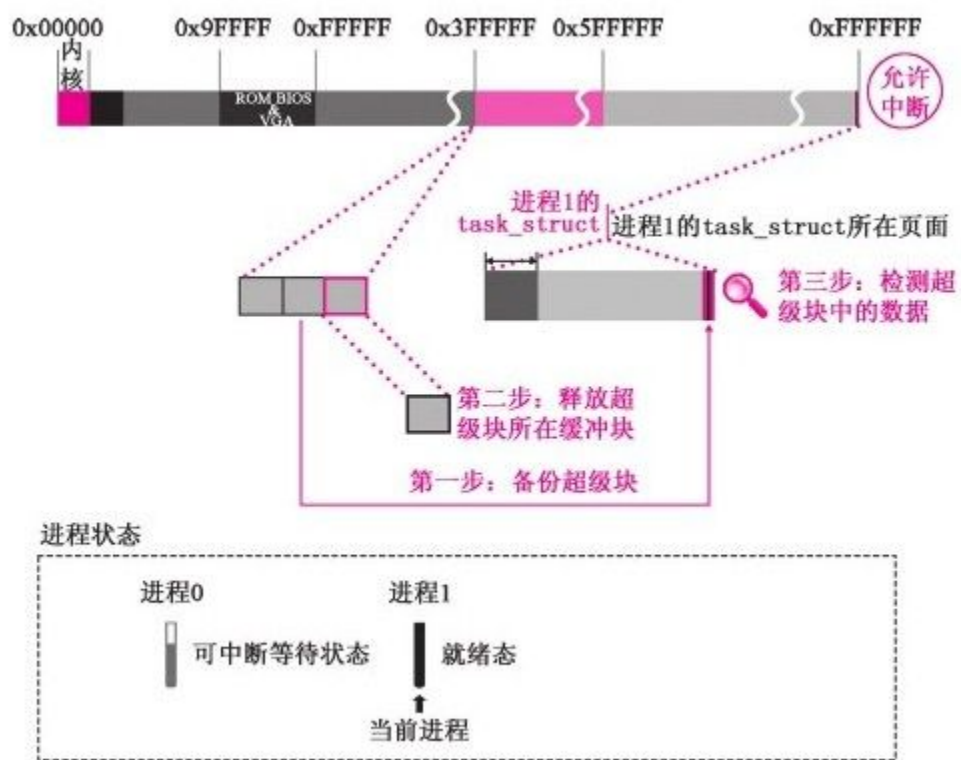


图 3-33 备份超级块并检测数据

接下来调用**breada**（）函数，把与文件系统相关的内容，从软盘上拷贝到虚拟盘中，然后及

时释放缓冲块，最终完成“格式化”这个过程，如图3-34所示。

复制结束后，将虚拟盘设置为根设备。

执行代码如下：

```
//代码路径： kernel/blk_dev/ramdisk.c:
```

```
void rd_load (void)
```

```
{
```

```
.....
```

```
printk ("Loading%d bytes into ram disk.....0000k",
```

```
nblocks << BLOCK_SIZE_BITS) ;
```

```
cp=rd_start;
```

```
while (nblocks) { //将软盘上准备格式化用的根文件系统复制到虚  
拟盘上
```

```
if (nblocks > 2)
```

```
bh=breada (ROOT_DEV,block,block+1, block+2, -1) ;
```

```

else

bh=bread (ROOT_DEV,block) ;

if (! bh) {

printk ("I/O error on block%d,aborting load\n",

block) ;

return;

}

(void) memcpy (cp,bh->b_data,BLOCK_SIZE) ;

brelse (bh) ;

printk ("\010\010\010\010\010%4dk", i) ;

cp+=BLOCK_SIZE;

block++;

nblocks--;

i++;

}

printk ("\010\010\010\010\010done\n") ;

ROOT_DEV=0x0101; //设置虚拟盘为根设备

```

}

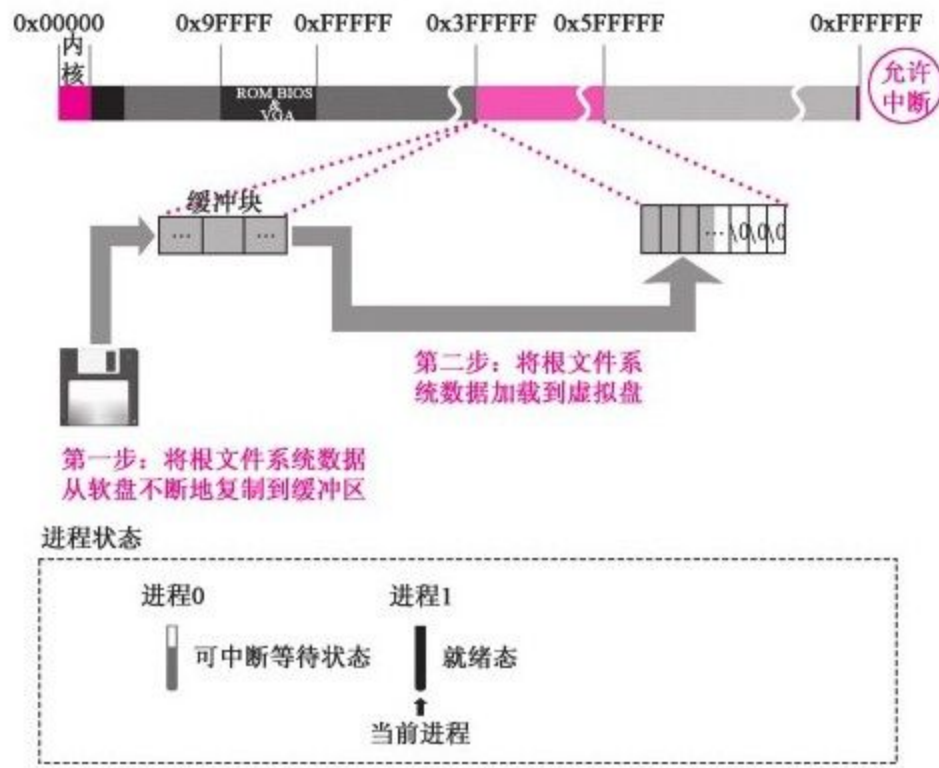


图 3-34 将根文件系统从软盘复制到虚拟盘

下面将要介绍在虚拟盘这个根设备上加载根文件系统。

3.3.3 进程1在根设备上加载根文件系统

操作系统中加载根文件系统涉及文件、文件系统、根文件系统、加载文件系统、加载根文件系统这几个概念。为了更容易理解，这里我们只讨论块设备，也就是软盘、硬盘、虚拟盘（有关块设备的详细讨论请阅读第5、7章）。

操作系统中的文件系统可以大致分成两部分；一部分在操作系统内核中，另一部分在硬盘、软盘、虚拟盘中。

文件系统是用来管理文件的。文件系统用i节点来管理文件，一个i节点管理一个文件，i节点和文件一一对应。文件的路径在操作系统中由目录文件中的目录项管理，一个目录项对应一级路

径，目录文件也是文件，也由i节点管理。一个文件挂在一个目录文件的目录项上，这个目录文件根据实际路径的不同，又可能挂在另一个目录文件的目录项上。一个目录文件有多个目录项，可以形成不同的路径。效果如图3-35所示。

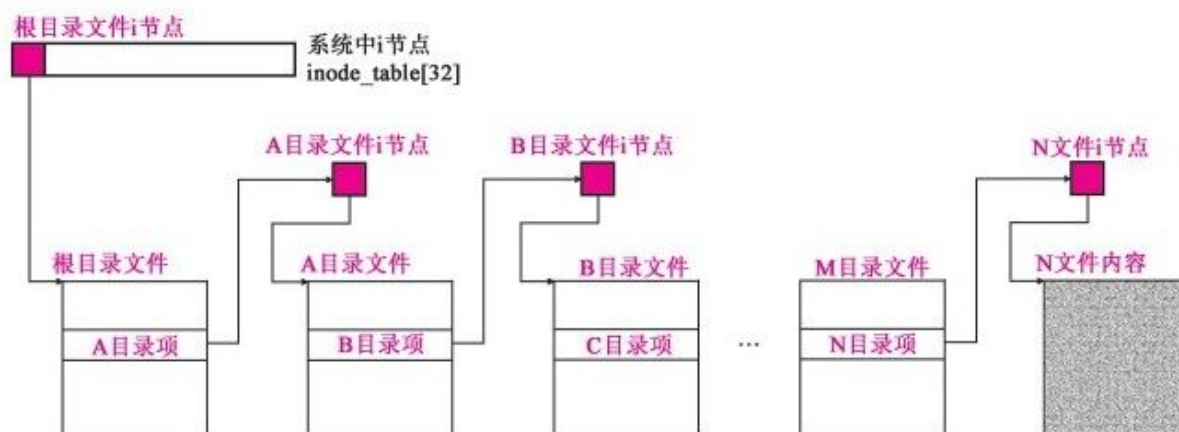


图 3-35 文件路径与i节点关系示意图

所有的文件（包括目录文件）的i节点最终挂接成一个树形结构，树根i节点就叫这个文件系统的根i节点。一个逻辑设备（一个物理设备可以分成多个逻辑设备，比如物理硬盘可以分成多个逻

辑硬盘) 只有一个文件系统, 一个文件系统只能包含一个这样的树形结构, 也就是说, 一个逻辑设备只能有一个根i节点。

加载文件系统最重要的标志, 就是把一个逻辑设备上的文件系统的根i节点, 关联到另一个文件系统的i节点上。具体是哪一个i节点, 由操作系统的使用者通过mount命令决定。

逻辑效果如图3-36所示。

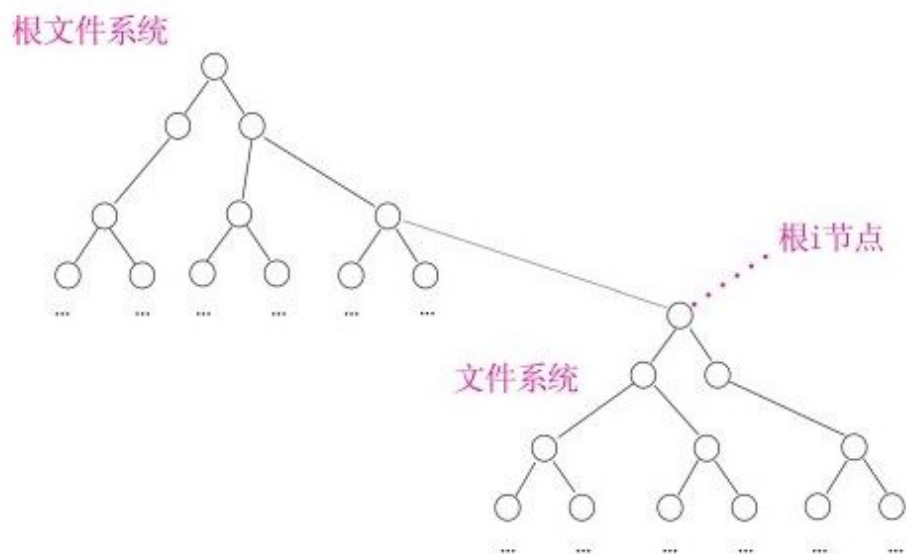


图 3-36 加载根文件系统逻辑效果图

另外，一个文件系统必须挂接在另一个文件系统上，按照这个设计，一定存在一个只被其他文件系统挂接的文件系统，这个文件系统就叫根文件系统，根文件系统所在的设备就叫根设备。

别的文件系统可以挂在根文件系统上，根文件系统挂在哪儿呢？

挂在`super_block[8]`上。

Linux 0.11操作系统中只有一个`super_block[8]`，每个数组元素是一个超级块，一个超级块管理一个逻辑设备，也就是说操作系统最多只能管理8个逻辑设备，其中只有一个根设备。加载根文件系统最重要的标志就是把根文件

系统的根i节点挂在super_block[8]中根设备对应的超级块上。

可以说，加载根文件系统有三个主要步骤：

- 1) 复制根设备的超级块到super_block[8]中，将根设备中的根i节点挂在super_block[8]中对应根设备的超级块上。

- 2) 将驻留缓冲区中16个缓冲块的根设备逻辑块位图、i节点位图分别挂接在super_block[8]中根设备超级块的s_zmap[8]、s_imap[8]上。

- 3) 将当前进程的pwd、root指针指向根设备的根i节点。

加载根文件系统和安装硬盘文件系统完成后的总体效果如图3-37所示。

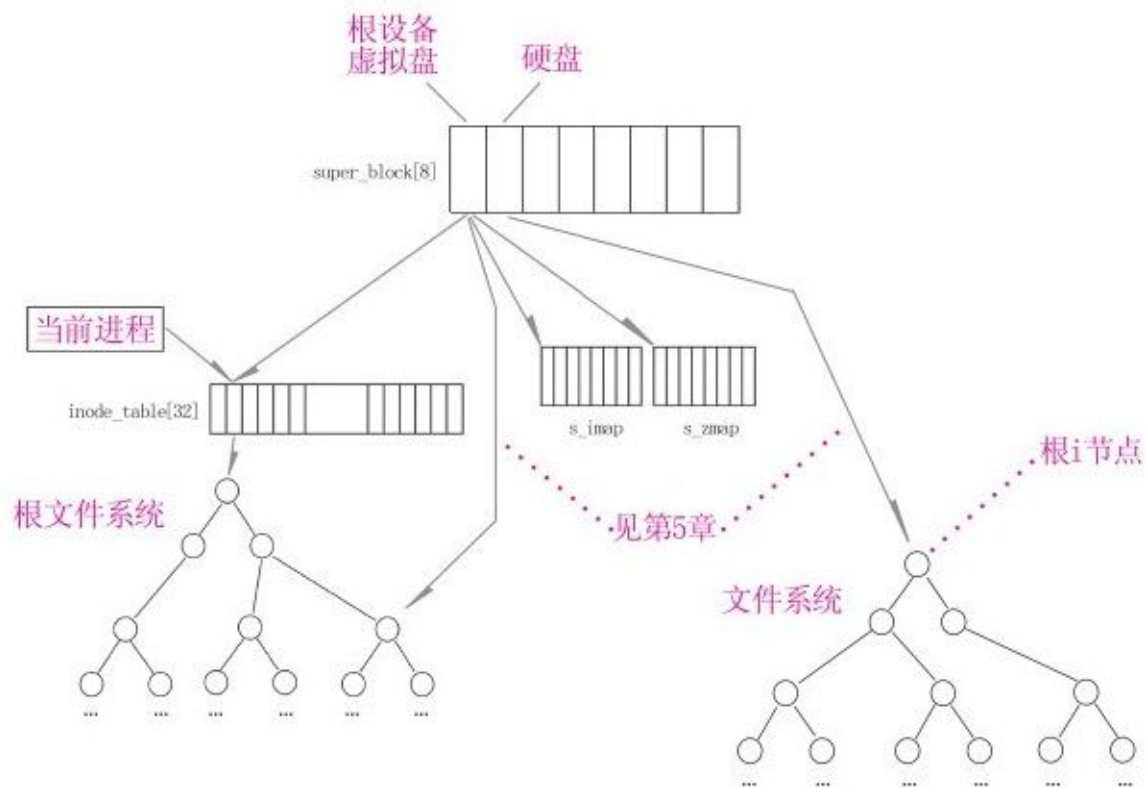


图 3-37 总体效果图

进程1通过调用`mount_root ()`函数实现在根设备虚拟盘上加载根文件系统。执行代码如下：

//代码路径: `kernel/blk_dev/hd.c`:

```
int sys_setup (void * BIOS)
```

```
{
```

```
.....

brelse (bh) ;

}

if (NR_HD)

    printk ("Partition table%s ok.\n\r", (NR_HD > 1) ? "s": "") ;

rd_load () ;

mount_root () ; //加载根文件系统

return (0) ;

}
```

1.复制根设备的超级块到super_block[8]中

进入mount_root () 函数后，初始化内存中的超级块super_block[8]，将每一项所对应的设备号加锁标志和等待它解锁的进程全部设置为0。系统只要想和任何一个设备以文件的形式进行数据交互，就要将这个设备的超级块存储在

super_block[8]中，这样可以通过super_block[8]获取这个设备中文件系统的最基本信息，根设备中的超级块也不例外，如图3-38所示。

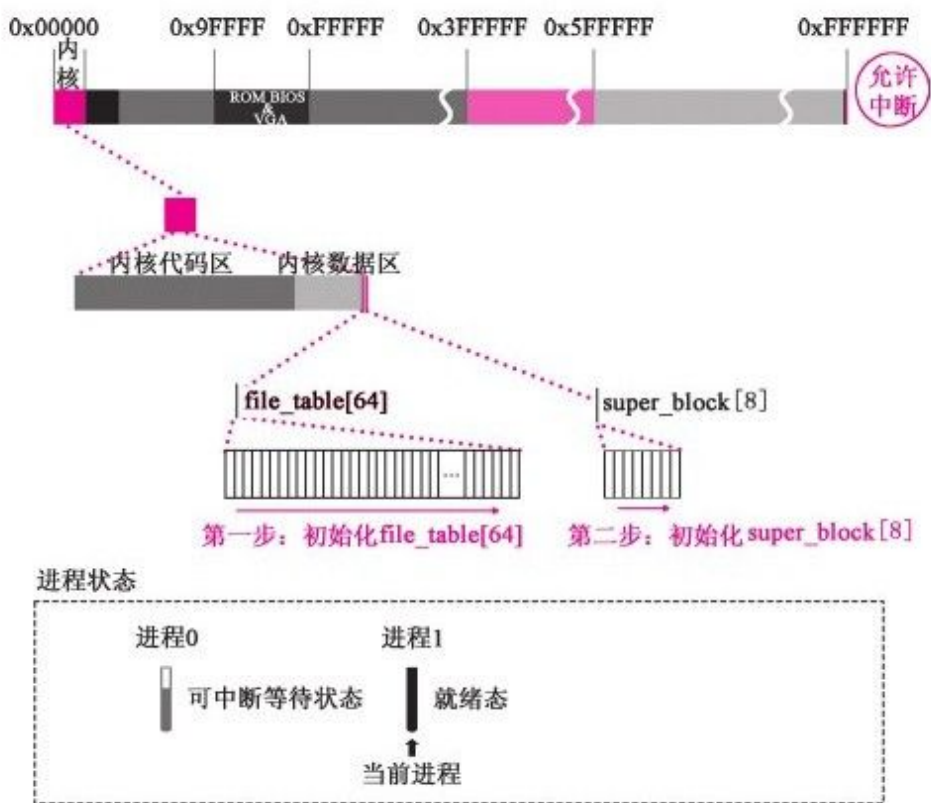


图 3-38 初始化file_table[64]和super_block[8]

执行代码如下：

```
//代码路径: fs/super.c:
```

```

void mount_root (void)

{

int i,free;

struct super_block * p;

struct m_inode * mi;

if (32 != sizeof (struct d_inode) )

panic ("bad i-node size") ;

for (i=0; i<NR_FILE; i++) //初始化file_table[64], 为后续程序
做准备

file_table[i].f_count=0;

if (MAJOR (ROOT_DEV) ==2) { //2代表软盘, 此时根设备是虚
拟盘, 是1。反之, 没有虚拟盘, 则加载软盘的根文件系统

printk ("Insert root floppy and press ENTER") ;

wait_for_keypress ( ) ;

}

//初始化super_block[8]

for (p=&super_block[0]; p<&super_block[NR_SUPER]; p++) {

p->s_dev=0;

```



```
p->s_lock=0;

p->s_wait=NULL;

}

if ( ! (p=read_super (ROOT_DEV) ) )

panic ("Unable to mount root") ;

.....

}
```

前面的rd_load () 函数已经“格式化”好虚拟盘，并设置为根设备。接下来调用read_super () 函数，从虚拟盘中读取根设备的超级块，复制到super_block[8]中。

执行代码如下：

```
//代码路径： fs/super.c:

void mount_root (void)
```

```
{  
  
.....  
  
if (! (p=read_super (ROOT_DEV) ) )  
  
panic ("Unable to mount root") ;  
  
.....  
  
}
```

在read_super () 函数中，先检测这个超级块是不是已经被读进super_block[8]中了。如果已经被读进来了，则直接使用，不需要再加载一次了。这与本章3.3.1节中先通过哈希表来检测缓冲块是否已经存在的道理是一样的。

执行代码如下：

```
//代码路径： fs/super.c:  
  
static struct super_block * read_super (int dev)
```

```
{  
  
    struct super_block * s;  
  
    struct buffer_head * bh;  
  
    int i,block;  
  
    if (! dev)  
  
        return NULL;  
  
    check_disk_change (dev) ; //检查是否换过盘， 并做相应处理  
  
    if (s=get_super (dev) )  
  
        return s;  
  
    .....  
  
}
```

因为此前没有加载过根文件系统，所以要在 `super_block[8]` 中申请一项。从图3-39中可以看出，此时找到的是 `super_block[8]` 结构中的第一

项。然后进行初始化并加锁，准备把根设备的超级块读出。

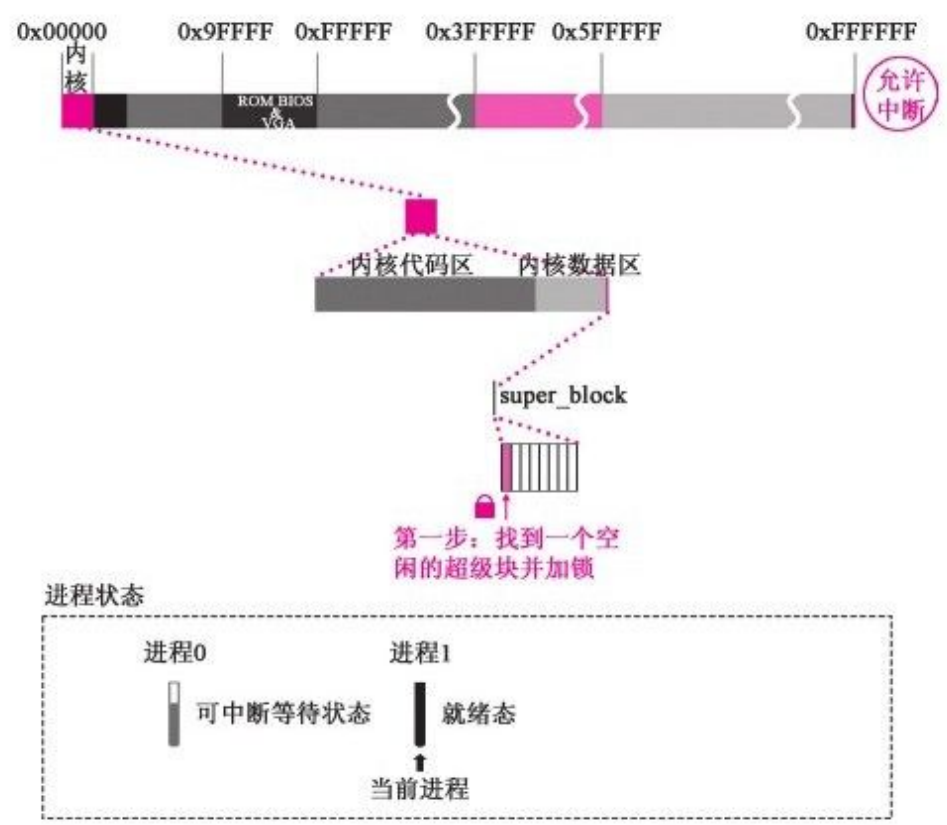


图 3-39 加载根文件系统超级块

对应的代码如下：

```
//代码路径： fs/super.c:

static struct super_block * read_super (int dev)
```

```

{

.....

for (s=0+super_block; s++) {

if (s >= NR_SUPER+super_block) //NR_SUPER是8

return NULL;

if (! s->s_dev)

break;

}

s->s_dev=dev;

s->s_isup=NULL;

s->s_imount=NULL;

s->s_time=0;

s->s_rd_only=0;

s->s_dirt=0;

lock_super (s) ; //锁定超级块

.....

}

```

调用**bread**（）函数，把超级块从虚拟盘上读进缓冲区，并从缓冲区复制到**super_block[8]**的第一项。**bread**（）函数在3.3.1节中已经说明。这里有一点区别，在3.3.1节中提到，如果给硬盘发送操作命令，则调用**do_hd_request**（）函数，而此时操作的是虚拟盘，所以要调用**do_rd_request**（）函数。值得注意的是，虚拟盘虽然被视为外设，但它毕竟是内存里面一段空间，并不是实际的外设，所以，调用**do_rd_request**（）函数从虚拟盘上读取超级块，不会发生类似硬盘中断的情况。

超级块复制进缓冲块以后，将缓冲块中的超级块数据复制到**super_block[8]**的第一项。从现在起，虚拟盘这个根设备就由**super_block[8]**的第一

项来管理，之后调用**brelease**（）函数释放这个缓冲块，如图3-40所示。

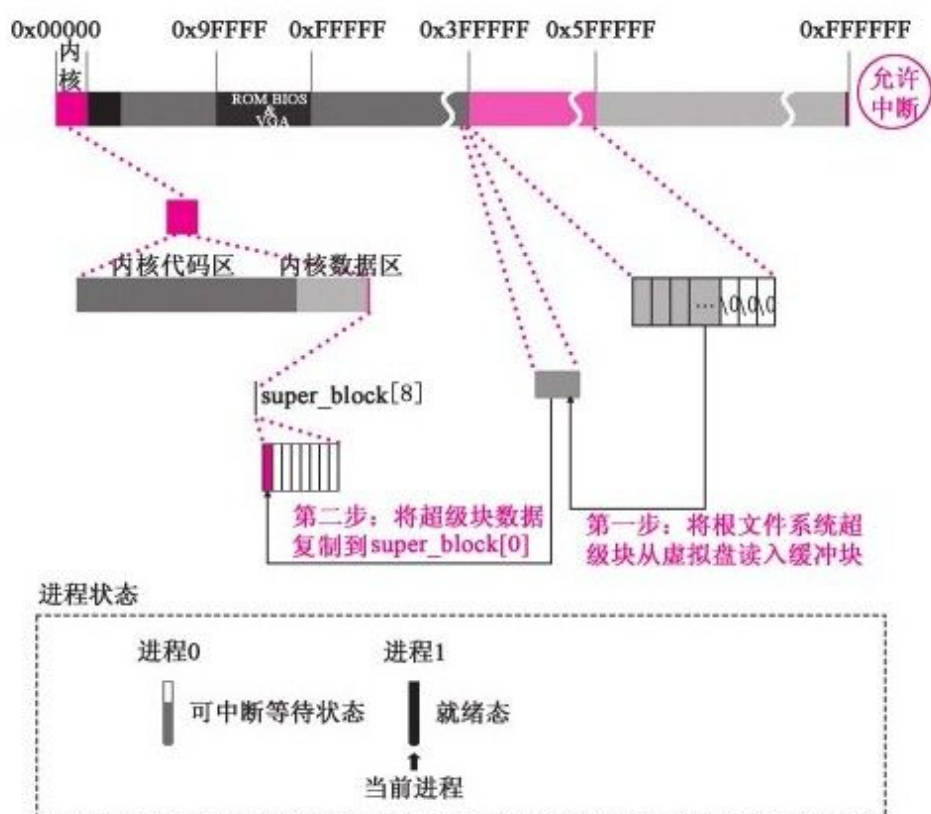


图 3-40 从虚拟盘读取超级块并复制到内核超级块表

执行代码如下：

```
//代码路径: fs/super.c:
```

```

static struct super_block * read_super (int dev)

{

.....

if (! (bh=bread (dev, 1) ) ) { //读根设备的超级块到缓冲区

s->s_dev=0;

free_super (s) ; //释放超级块

return NULL;

}

* ( (struct d_super_block *) s) = //将缓冲区中的超级块复制到

* ( (struct d_super_block *) bh->b_data) ; //super_block[8]第一
项

brelse (bh) ; //释放缓冲块

if (s->s_magic != SUPER_MAGIC) { //判断超级块的魔数
(SUPER_MAGIC) 是否正确

s->s_dev=0;

free_super (s) ; //释放超级块

return NULL;

}

```



```
.....
```

```
}
```

初始化super_block[8]中的虚拟盘超级块中的i节点位图s_imap、逻辑块位图s_zmap，并把虚拟盘上i节点位图、逻辑块位图所占用的所有逻辑块读到缓冲区，将这些缓冲块分别挂接到s_imap[8]和s_zmap[8]上。由于对它们的操作会比较频繁，所以这些占用的缓冲块并不被释放，它们将常驻在缓冲区内。

如图3-41所示，超级块通过指针与s_imap和s_zmap实现挂接。

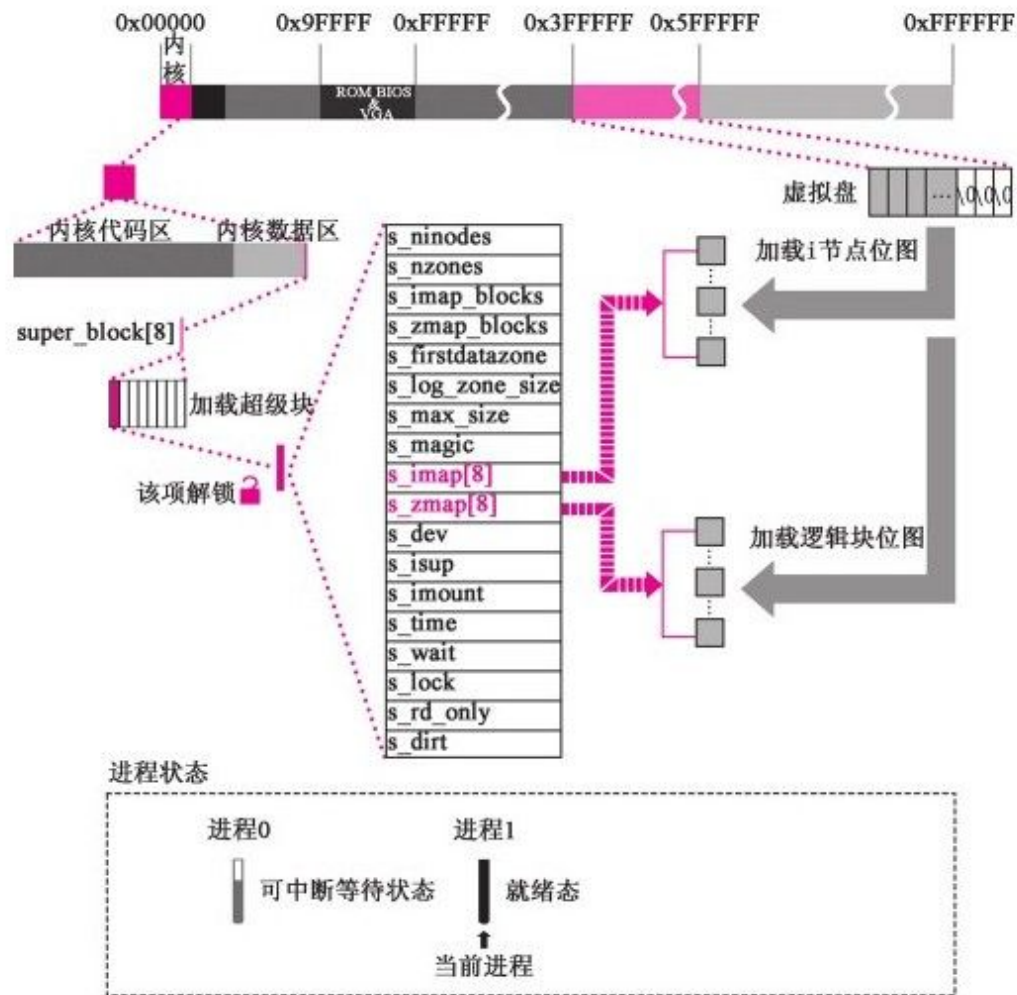


图 3-41 读取逻辑块位图和i节点位图

执行代码如下：

//代码路径： fs/super.c:

```
static struct super_block * read_super (int dev)
```

```
{
```

.....

```
for (i=0; i<I_MAP_SLOTS; i++) //初始化s_imap[8]、s_zmap[8]
```

```
s->s_imap[i]=NULL;
```

```
for (i=0; i<Z_MAP_SLOTS; i++)
```

```
s->s_zmap[i]=NULL;
```

block=2; //虚拟盘的第一块是超级块，第二块开始是i节点位图和逻辑块位图

```
for (i=0; i<s->s_imap_blocks; i++) //把虚拟盘上i节点位图所占用的所有逻辑块
```

```
if (s->s_imap[i]=bread (dev,block) ) //读到缓冲区，分别挂接到s_imap[8]上
```

```
block++;
```

```
else
```

```
break;
```

```
for (i=0; i<s->s_zmap_blocks; i++) //把虚拟盘上逻辑块位图所占用的所有逻辑块
```

```
if (s->s_zmap[i]=bread (dev,block) ) //读到缓冲区，分别挂接到s_zmap[8]上
```

```
block++;
```

```
else
```

```
break;
```

```
if (block != 2 + s->s_imap_blocks + s->s_zmap_blocks) { //如果i节点位图、逻辑块位
```

```
for (i=0; i<I_MAP_SLOTS; i++) //图所占用的块数不对，说明操作系统有问题，则应释放前
```

```
brelse (s->s_imap[i]) ; //面获得的缓冲块及超级块
```

```
for (i=0; i<Z_MAP_SLOTS; i++)
```

```
brelse (s->s_zmap[i]) ;
```

```
s->s_dev=0;
```

```
free_super (s) ;
```

```
return NULL;
```

```
} s->s_imap[0]->b_data[0]=1; //牺牲一个i节点，以防止查找算法返回0
```

```
s->s_zmap[0]->b_data[0]=1; //与0号i节点混淆
```

```
free_super (s) ;
```

```
return s;
```

```
}
```

2.将根设备中的根i节点挂在super_block[8]中
根设备超级块上

回到mount_root () 函数中，调用iget () 函数，从虚拟盘上读取根i节点。根i节点的意义在于，通过它可以到文件系统中任何指定的i节点，也就是能找到任何指定的文件。

执行代码如下：

```
//代码路径: fs/super.c:

void mount_root (void)

{

.....

if (! (p=read_super (ROOT_DEV) ) )

panic ("Unable to mount root") ;

if (! (mi=iget (ROOT_DEV,ROOT_INO) ) )
```

```
panic ("Unable to read root i-node") ;
```

```
.....
```

```
}
```

进入iget () 函数后，操作系统从i节点表inode_table[32]中申请一个空闲的i节点位置

(inode_table[32]是操作系统用来控制同时打开不同文件的最大数)。此时应该是首个i节点。对这个i节点进行初始化设置，其中包括该i节点对应的设备号、该i节点的节点号.....图3-42中给出了根目录i节点在内核i节点表中的位置。

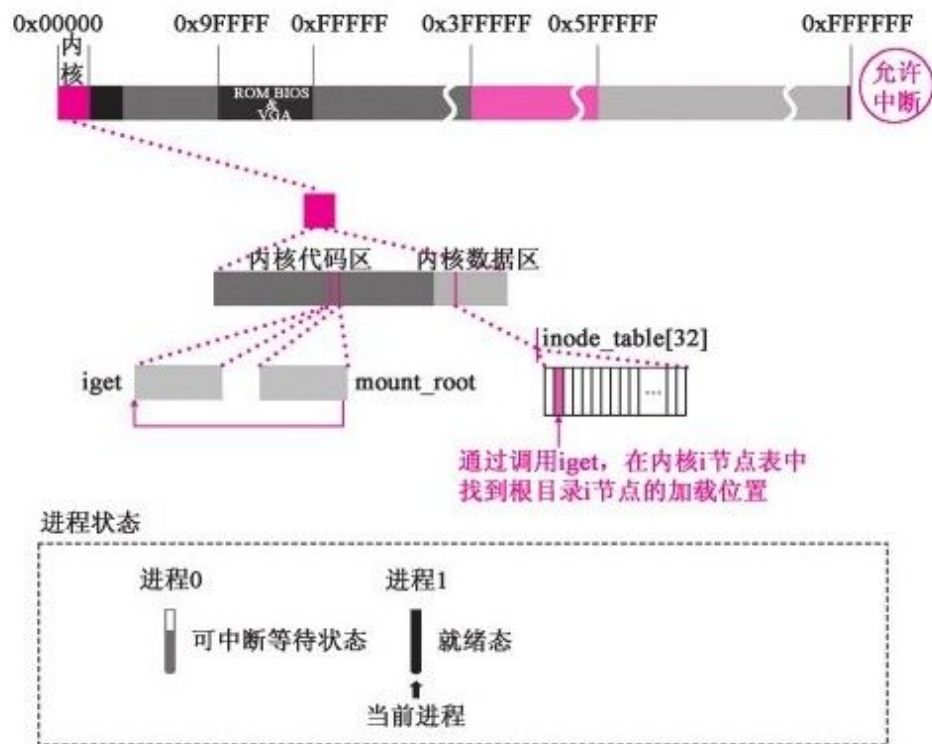


图 3-42 读取根目录i节点

对应代码如下：

//代码路径： fs/inode.c:

```
struct m_inode * iget (int dev,int nr)
{
    struct m_inode * inode, *empty;

    if (! dev)
```

```
panic ("iget with dev==0") ;
```

```
empty=get_empty_inode () ; //从inode_table[32]中申请一个空闲的  
i节点
```

```
inode=inode_table;
```

```
while (inode<NR_INODE+inode_table) { //查找与参数相同的  
inode
```

```
if (inode->i_dev !=dev||inode->i_num !=nr) {
```

```
inode++;
```

```
continue;
```

```
}
```

```
wait_on_inode (inode) ; //等待解锁
```

```
if (inode->i_dev !=dev||inode->i_num !=nr) { //如等待期间发生
```

```
inode=inode_table; //变化，继续查找
```

```
continue;
```

```
}
```

```
inode->i_count++;
```

```
if (inode->i_mount) {
```

```
int i;
```


for (i=0; i<NR_SUPER; i++) //如是mount点, 则查找对应的超级块

if (super_block[i].s_imount==inode)

break;

if (i>=NR_SUPER) {

printk ("Mounted inode hasn't got sb\n") ;

if (empty)

iput (empty) ;

return inode;

}

iput (inode) ;

dev=super_block[i].s_dev; //从超级块中获取设备号

nr=ROOT_INO; //ROOT_INO为1, 根i节点号

inode=inode_table;

continue;

}

if (empty)

iput (empty) ;

```
return inode;

}

if (! empty)

return (NULL) ;

inode=empty;

inode->i_dev=dev; //初始化

inode->i_num=nr;

read_inode (inode) ; //从虚拟盘上读出根i节点

return inode;

}
```

在read_inode () 函数中，先给inode_table[32]中的这个i节点加锁。在解锁之前，这个i节点就不会被别的程序占用。之后，通过该i节点所在的超级块，间接地计算出i节点所在的逻辑块号，并将i节点所在的逻辑块整体读出，从中提取这个i节点

的信息，载入刚才加锁的i节点位置上，如图3-43所示，注意inode_table[32]中的变化。最后，释放缓冲块并将锁定的i节点解锁。

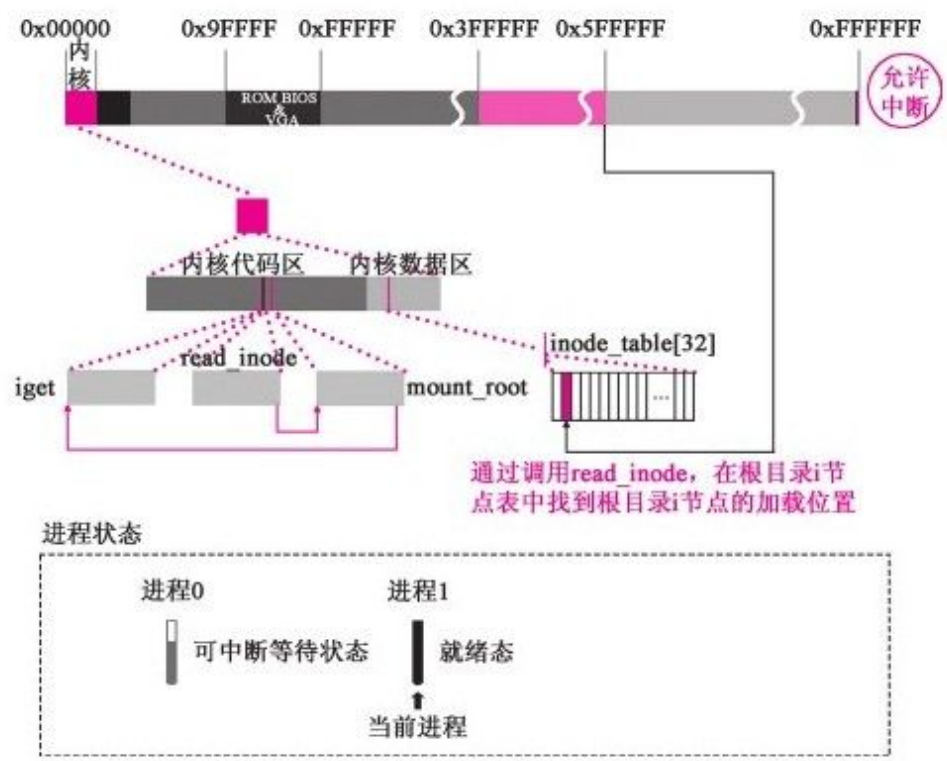


图 3-43 读取i节点

执行的代码如下：

//代码路径： fs/inode.c:

```

static void read_inode (struct m_inode * inode)

{

.....

lock_inode (inode) ; //锁定inode

if ( ! (sb=get_super (inode->i_dev) ) ) //获得inode所在设备的
超级块

.....

block=2+sb->s_imap_blocks+sb->s_zmap_blocks+

(inode->i_num-1) /INODES_PER_BLOCK;

if ( ! (bh=bread (inode->i_dev,block) ) ) //读inode所在逻辑块
进缓冲块

panic ("unable to read i-node block") ;

* (struct d_inode *) inode=//整体复制

( (struct d_inode *) bh->b_data)

[ (inode->i_num-1) %INODES_PER_BLOCK];

brelse (bh) ; //释放缓冲块

unlock_inode (inode) ; //解锁

}

```

回到iget () 函数，将inode指针返回给mount_root () 函数，并赋给mi指针。

下面是加载根文件系统的标志性动作：

将inode_table[32]中代表虚拟盘根i节点的项挂接到super_block[8]中代表根设备虚拟盘的项中的s_isup、s_imount指针上。这样，操作系统在根设备上可以通过这里建立的关系，一步步地把文件找到。

3.将根文件系统与进程1关联

对进程1的task_struct中与文件系统i节点有关的字段进行设置，将根i节点与当前进程（现在就是进程1）关联起来，如图3-44所示。

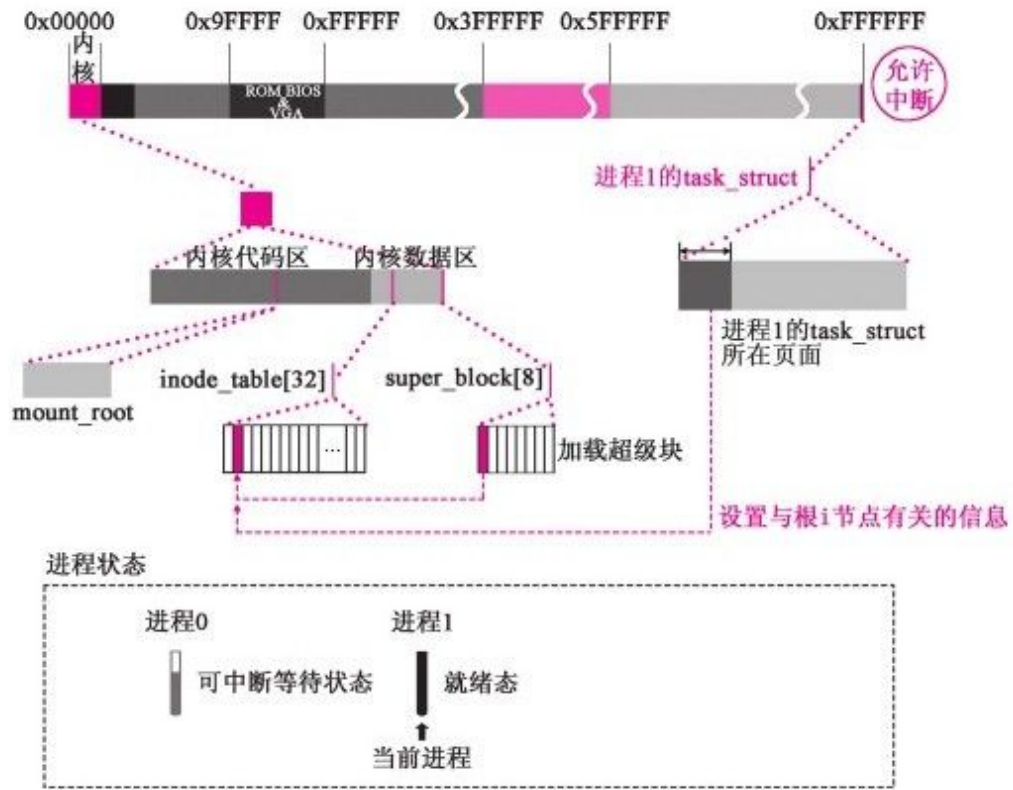


图 3-44 加载根文件系统完成并返回

执行代码如下：

//代码路径： fs/super.c:

```
void mount_root (void)
```

```
{
```

```
.....
```

点

```
if (! (mi=iget (ROOT_DEV,ROOT_INO) ) ) //根设备的根i节  
panic ("Unable to read root i-node" ) ;  
  
mi->i_count+=3; /*NOTE! it is logically used 4 times,not 1*/  
  
p->s_isup=p->s_imount=mi; //标志性的一步!  
  
current->pwd=mi; //当前进程（进程1）掌控根文件系统的根i节  
点,  
  
current->root=mi; //父子进程创建机制将这个特性遗传给子进程  
  
.....  
  
}
```

得到了根文件系统的超级块，就可以根据超级块中“逻辑块位图”里记载的信息，计算出虚拟盘上数据块的占用与空闲情况，并将此信息记录在3.3.3节中提到的驻留在缓冲区中“装载逻辑块位图信息的缓冲块中”。执行代码如下：

//代码路径： fs/super.c:

```

void mount_root (void)

{

.....

free=0;

i=p->s_nzones;

while (--i >= 0) //计算虚拟盘中空闲逻辑块的总数

if (! set_bit (i&8191, p->s_zmap[i >> 13]->b_data) )

free++;

printf ("%d/%d free blocks\n\r", free,p->s_nzones) ;

free=0;

i=p->s_ninodes+1;

while (--i >= 0) //计算虚拟盘中空闲的i节点的总数

if (! set_bit (i&8191, p->s_imap[i >> 13]->b_data) )

free++;

printf ("%d/%d free inodes\n\r", free,p->s_ninodes) ;

}

```

到此为止，`sys_setup`（）函数就全都执行完毕了。因为这个函数也是由于产生软中断才被调用的，所以返回`system_call`中执行，之后会执行`ret_from_sys_call`。这时候的当前进程是进程1，所以下面将调用`do_signal`（）函数（只要当前进程不是进程0，就要执行到这里），对当前进程的信号位图进行检测，执行代码如下：

```
//代码路径： kernel/system_call.s:
```

```
.....
```

```
ret_from_sys_call:
```

```
movl _current, %eax#task[0]cannot have signals
```

```
cmpl _task, %eax
```

```
je 3f
```

```
cmpw $0x0f,CS (%esp) #was old code segment supervisor?
```

```
jne 3f
```

```
cmpw $0x17, OLDSS (%esp) #was stack segment=0x17?
```

```
jne 3f
```

```
movl signal (%eax) , %ebx#下面是取信号位图.....
```

```
movl blocked (%eax) , %ecx
```

```
notl %ecx
```

```
andl %ebx, %ecx
```

```
bsfl %ecx, %ecx
```

```
je 3f
```

```
btrl %ecx, %ebx
```

```
movl %ebx,signal (%eax)
```

```
incl %ecx
```

```
pushl %ecx
```

```
call _do_signal#调用do_signal ()
```

```
.....
```

现在，当前进程（进程1）并没有接收到信号，调用do_signal（）函数并没有实际的意义。

至此，`sys_setup`（）的系统调用结束，进程1将返回3.3节中讲到的代码的调用点，准备下面代码的执行。

//代码路径：init/main.c:

```
void init (void)
```

```
{
```

```
.....
```

```
int pid,i;
```

```
setup ( (void *) &drive_info) ;
```

```
(void) open ("/dev/tty0", O_RDWR, 0) ;
```

```
(void) dup (0) ;
```

```
(void) dup (0) ;
```

```
printf ("%d buffers=%d bytes buffer space\n\r", NR_BUFFERS,
```

```
NR_BUFFERS * BLOCK_SIZE) ;
```

```
.....
```

```
}
```

至此，进程0创建进程1，进程1为安装硬盘文件系统做准备、“格式化”虚拟盘并用虚拟盘取代软盘为根设备、在虚拟盘上加载根文件系统的内容讲解完毕。

3.4 本章小结

本章详细讲解了进程0创建进程1的全过程。后续所有进程的创建过程与这个过程基本相同。透彻理解这个创建过程，为理解后续的进程创建打下坚实的基础。

本章还讲解了操作系统启动以来内核做的第一次进程调度，内容涉及了进程调度的很多代码，为更深入地理解进程调度起到了很好的铺垫作用。

最后，本章详细讲解了进程1第一次执行后所做的设置硬盘信息、格式化虚拟盘、加载根文件系统等工作。

第4章 进程2的创建及执行

现在，计算机中已经创建了两个进程：进程0、进程1。本章我们将要详细讲解进程1创建进程2的过程，以及进程2的执行，最终shell进程开始执行，整个boot工作完成，实现系统怠速。

4.1 打开终端设备文件及复制文件句柄

shell进程是用户界面进程。计算机用户使用显示器、键盘（终端设备）通过shell进程与操作系统之间进行人机交互。

4.1.1 打开标准输入设备文件

tty0文件加载后，就形成了如图4-1所示的效果图。

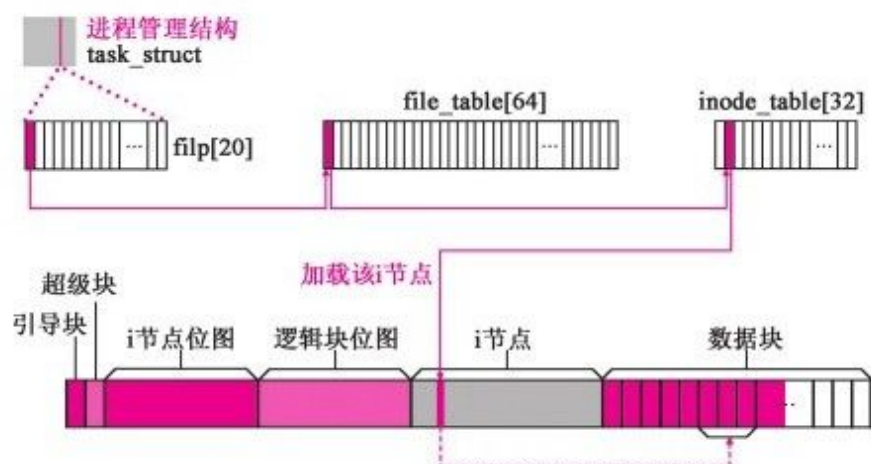


图 4-1 打开tty0文件后，文件信息在内存和进程中的分布图

1.file_table[0]挂接在进程1的filp[0]

在加载完根文件系统之后，进程1在其支持下，通过调用open（）函数来打开标准输入设备文件，执行代码如下：

//代码路径: init/main.c:

```
void init (void)
```

```
{
```

```
int pid,i;
```

```
setup ( (void *) &drive_info) ;
```

```
    (void) open ("/dev/tty0", O_RDWR, 0) ; //创建标准输入设备，其中/dev/tty0是该文件的路径名
```

```
    (void) dup (0) ; //创建标准输出设备
```

```
    (void) dup (0) ; //创建标准错误输出设备
```

```
    printf ("%d buffers=%d bytes buffer space\n\r", NR_BUFFERS, //  
在标准输出设备支持下，显示信息
```

```
NR_BUFFERS * BLOCK_SIZE) ;
```

```
    printf ("Free mem: %d bytes\n\r", memory_end-  
main_memory_start) ;
```

```
.....
```

```
}
```

`open`（）函数执行后产生软中断，并最终映射到内核中`sys_open`（）函数去执行。此映射过程与第3章3.1.1节中`fork`（）函数映射到`sys_fork`（）函数的技术路线大体一致。执行代码如下：

```
//代码路径： fs/open.c:
```

```
int open（const char * filename,int flag, .....）
```

```
{
```

```
register int res;
```

```
va _list arg;
```

```
va _start（arg,flag）；
```

```
__asm__（"int$0x80"//以下代码与fork到sys_fork的映射类似，详情  
参看第3章3.1.1节
```

```
： "=a"（res）
```

```
： "0"（__NR_open）， "b"（filename）， "c"（flag），
```

```
"d"（va_arg（arg,int）））；
```

```
if（res >=0）
```

```
return res;

errno=-res;

return-1;

}
```

进入sys_open () 函数，内核先将进程1的 filp[20]与file_table[64]挂接，建立进程与file_table[64]的关系，执行代码如下：

//代码路径： fs/open.c:

```
int sys_open (const char * filename,int flag,int mode)

{

    struct m_inode * inode;

    struct file * f;

    int i,fd;

    mode&=0777&~current->umask;

    for (fd=0; fd<NR_OPEN; fd++) //遍历进程1的filp
```

if (! current-> flp[fd]) //直到获取一个空闲项，fd就是这个空闲项的项号

break;

if (fd >= NR_OPEN) //如果此条件成立，说明filp[20]中没有空闲项了，直接返回

return-EINVAL;

current-> close_on_exec &= ~ (1 << fd) ;

f=0+file_table; //获取file_table[64]首地址

for (i=0; i<NR_FILE; i++, f++) //遍历file_table[64]

if (! f-> f_count) break; //直到获取一个空闲项，f就是这个空闲项的指针

if (i >= NR_FILE) //如果此条件成立，说明file_table[64]中没有空闲项了，

//直接返回

return-EINVAL;

(current-> filp[fd]=f) -> f_count++; //将进程1的filp[20]与file_table[64]挂接，并增加引用计数

if ((i=open_namei (filename,fag,mode, &inode)) < 0) { //获取文件i节点

current-> filp[fd]=NULL;

```
f->f_count=0;

return i;

}

.....

}
```

挂接情景如图4-2所示。

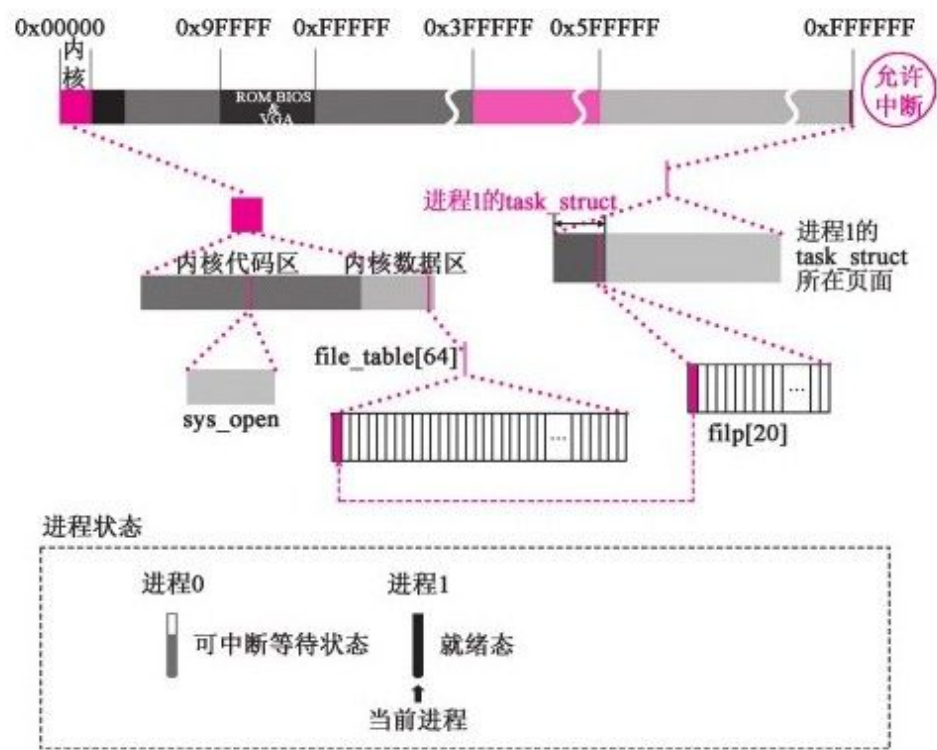


图 4-2 打开终端设备文件的准备工作

2.确定绝对路径起点

内核将调用`open_namei ()` 函数，最终获取标准输入设备文件的i节点，具体执行代码如下：

//代码路径： fs/open.c:

```
int sys_open (const char * filename,int flag,int mode)

{

    struct m_inode * inode;

    struct file * f;

    int i,fd;

    mode&=0777&~current->umask;

    for (fd=0; fd<NR_OPEN; fd++)

        if (! current->flp[fd])

            break;

    if (fd>=NR_OPEN)

        return-EINVAL;
```

```
current->close_on_exec&=~ (1<<fd) ;
```

```
f=0+file_table;
```

```
for (i=0; i<NR_FILE; i++, f++)
```

```
if (! f->f_count) break;
```

```
if (i>=NR_FILE)
```

```
return-EINVAL;
```

```
(current->filp[fd]=f) -> f_count++;
```

```
if ( (i=open_namei (filename,flag,mode, &inode) ) < 0) {//此时  
的filename就是路径/dev/tty0的指针
```

```
current->filp[fd]=NULL;
```

```
f->f_count=0;
```

```
return i;
```

```
}
```

```
.....
```

```
}
```

这一目标是通过不断分析路径名来实现的。分析工作的第一阶段是调用`dir_namei`（）函数，获取枝梢`i`节点，即`/dev/tty0`路径中`dev`目录文件的`i`节点；第二阶段是调用`find_entry`（）函数，通过此`i`节点，找到`dev`目录文件中`tty0`这一目录项，再通过该目录项找到`tty0`文件的`i`节点。

第一阶段，调用`dir_namei`（）函数，具体执行代码如下：

```
//代码路径： fs/namei.c:

int open_namei (const char * pathname,int flag,int mode, //pathname
就是路径/dev/tty0的指针

struct m_inode ** res_inode)

{

const char * basename; //basename记录目录项名字前面'/'的地址

int inr,dev,namelen; //namelen记录名字的长度
```

```

struct m_inode * dir, *inode;

struct buffer_head * bh;

struct dir_entry * de; //de用来指向目录项内容

if ( (flag&O_TRUNC) && ! (flag&O_ACCMODE) )

flag|=O_WRONLY;

mode&=0777&~current->umask;

mode|=I_REGULAR;

if ( ! (dir=dir_namei (pathname, &namelen, &
basename) ) ) //获取枝梢i节点

return-ENOENT;

if ( ! namelen ) { /*special case: '/usr/etc*/

if ( ! (flag& (O_ACCMODE|O_CREAT|O_TRUNC) ) ) {

*res_inode=dir;

return 0;

}

iput (dir) ;

return-EISDIR;

}

```



```
    bh=find_entry (&dir,basename,namelen, &de) ; //通过枝梢i节点，找到目标文件的目录项
```

```
    .....
```

```
}
```

`dir_namei ()` 函数中将首先调用`get_dir ()` 函数来获取枝梢i节点，之后再通过解析路径名，获取tty0目录项的地址和文件名长度信息。调用`get_dir ()` 函数的具体执行代码如下：

//代码路径： fs/namei.c:

```
static struct m_inode * dir_namei (const char * pathname, //pathname  
就是路径/dev/tty0的指针
```

```
int * namelen,const char ** name)
```

```
{
```

```
char c;
```

```
const char * basename;
```

```
struct m_inode * dir;
```

```
if (! (dir=get_dir (pathname) ) ) //获取i节点的执行函数

return NULL;

basename=pathname;

while (c=get_fs_byte (pathname++)) //逐个遍历/dev/tty0字符串，每次循环都将一个字符复制给c，直到字符串结束

if (c=='/')

basename=pathname;

*namelen=pathname-basename-1; //确定tty0名字的长度

*name=basename; //得到tty0前面'/'字符的地址

return dir;

}
```

值得注意的是，`get_fs_byte ()` 函数是解析路径的核心函数，可以从路径中逐一提取字符串。该函数在后面具体的路径解析工作中还会用到，它的内部处理过程如下：

```

//代码路径: include/asm/Segment.h:

extern inline unsigned char get_fs_byte (const char * addr)

{

    unsigned register char _v;

    __asm__ ("movb%%fs: %1, %0"//movb指令可以将8位, 即1字节
数据移入指定寄存器 (fs)

: "=r" (_v) //v是输出的字符

: "m" (*addr) ) ; // *addr是输入的内存地址

    return _v;

}

```

`get_dir ()` 函数首先确定路径的绝对起点, 即分析`"/dev/tty0"`这个路径名的第一个字符是不是`'/'`。如果是`'/'`, 就确定这是绝对路径名, 因此将从根`i`节点开始查找文件。这个根`i`节点已在第3章3.3.3节中加载根文件系统时载入, 它被确定为路

径绝对起点，同时，它被引用，其引用计数也随之增加。这部分执行代码如下：

//代码路径： fs/namei.c:

static struct m_inode * get_dir (const char * pathname) //pathname就是路径/dev/tty0的指针

{

char c;

const char * thisname;

struct m_inode * inode;

struct buffer_head * bh;

int namelen,inr,idev;

struct dir_entry * de;

if (! current->root||! current->root->i_count) //当前进程的根i节点不存在或引用计数为0，死机

panic ("No root inode") ;

if (! current->pwd||! current->pwd->i_count)

//当前进程的当前工作目录根i节点不存在或引用计数为0，死机

```
panic ("No cwd inode") ;

if ( (c=get_fs_byte (pathname) ) == '/') { //此处识别
出"/dev/tty0"这个路径的第一个字符是 '/'

inode=current->root;

pathname++;

} else if I

inode=current->pwd;

else

return NULL; /*empty name is bad*/

inode->i_count++; //该i节点的引用计数也随之加1

.....

}
```

确定路径起点的情景如图4-3所示。

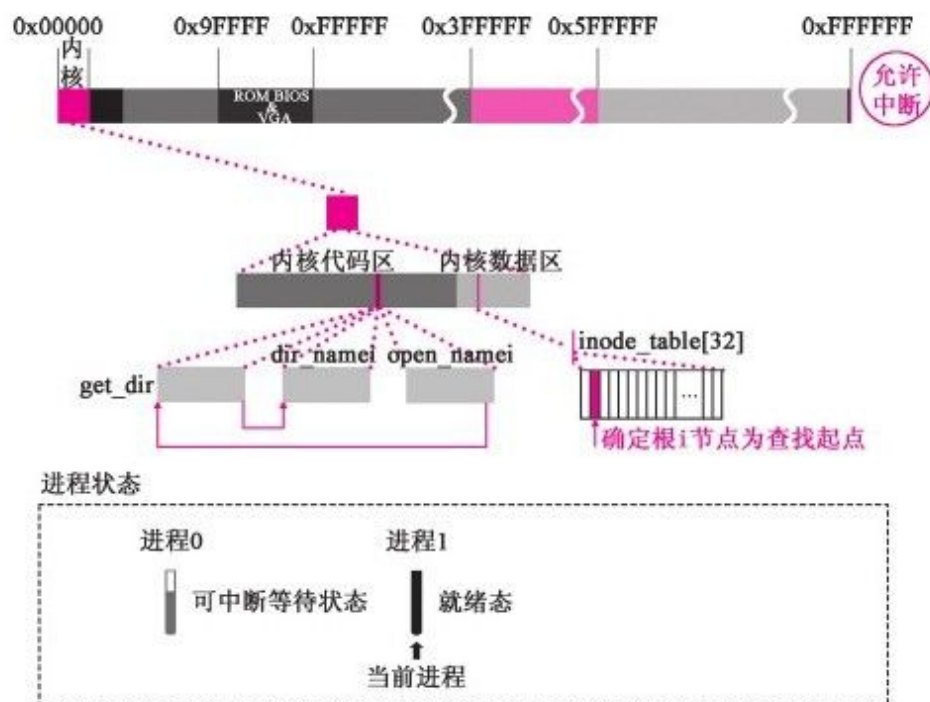


图 4-3 文件名解析准备工作

3. 获得dev目录文件i节点

从根i节点开始，遍历并解析"/dev/tty0"这个路径名，首先会解析到dev这个目录项，之后将在虚拟盘上找到这个目录项所在的逻辑块，并读进指定的缓冲块，具体解析过程如下：

//代码路径： fs/namei.c:

```

static struct m_inode * get_dir (const char * pathname)

{

char c;

const char * thisname; //thisname记录目录项名字前面'/'的地址

struct m_inode * inode;

struct buffer_head * bh;

int namelen,inr,idev; //namelen记录名字的长度

struct dir_entry * de; //de用来指向目录项内容

.....

if ( (c=get_fs_byte (pathname) ) =='/') {

inode=current->root;

    pathname++; //pathname原本是/dev/tty0这个字符串中第一个字符
    的指针，即指向'/', ++后指向'd'

} else if I

inode=current->pwd;

else

return NULL; /*empty name is bad*/

inode->i_count++;

```

```

while (1) { //循环以下过程，直到找到枝梢i节点为止

    thisname=pathname; //thisname也会指向'd'

    if (! S_ISDIR (inode->i_mode) || ! permission
        (inode,MAY_EXEC) ) {

        iput (inode) ;

        return NULL;

    }

    //每当检索到字符串中的'/'字符，或者c为'\0'，循环都会跳出

    for (namelen=0; (c=get_fs_byte (pathname++)) && (c!
        ='/') ; namelen++)

        /*nothing*/; //注意这个分号

    if (! c)

        return inode;

    //通过目录文件的i节点和目录项信息，获取目录项

    if (! (bh=find_entry (&inode,thisname,namelen, &de) ) ) {

        iput (inode) ;

        return NULL;

    }

```



```
inr=de->inode;

idev=inode->i_dev;

brelse (bh) ;

iput (inode) ;

if (! (inode=iget (idev,inr) ) )

return NULL;

}

}
```

`get_fs_byte ()` 函数再次被用到，从`/dev/tty0`路径名`dev`的'd'字符开始遍历，遇到'\n'后跳出循环，`namelen`数值累加为3。这些信息将和根`i`节点指针一起，作为`find_entry ()`函数的参数使用。`find_entry ()`函数会将目录所在的逻辑块读入缓冲块。

此处值得注意的是，上面代码中find_entry

() 函数最后一个参数de所指向的数据结构是目录项结构，代码如下：

```
//代码位置: /include/linux/fs.h

#define NAME_LEN 14

struct dir_entry{//目录项结构

    unsigned short inode; //目录项所对应目录文件在设备上的i节点号

    char name[NAME_LEN]; //目录项名字，14字节

};
```

得到了i节点号，就可以得到“dev”目录项所对应目录文件的i节点，内核就可以进而通过i节点找到dev目录文件。获取i节点的代码如下：

```
//代码路径: fs/namei.c:

static struct m_inode * get_dir (const char * pathname)
```

```

{

.....

if ( (c=get_fs_byte (pathname) ) == '/') {

inode=current->root;

pathname++;

}else if I

inode=current->pwd;

else

return NULL; /*empty name is bad*/

inode->i_count++;

while (1) { //循环以下过程，直到找到枝梢i节点为止

thisname=pathname;

if (! S_ISDIR (inode->i_mode) || ! permission
(inode,MAY_EXEC) ) {

iput (inode) ;

return NULL;

}

```

```

    for (namelen=0; (c=get_fs_byte (pathname++)) && (c!
    ='/') ; namelen++)

        /*nothing*/;

    if (! c)

        return inode;

    if (! (bh=find_entry (&inode,thisname,namelen, &de) ))
    { //de会指向dev目录项

        iput (inode) ;

        return NULL;

    }

    inr=de->inode; //通过目录项找到i节点号

    idev=inode->i_dev; //注意，这个inode是根i节点，这里通过根i节
    点找到设备号

    brelse (bh) ;

    iput (inode) ;

    if (! (inode=iget (idev,inr) )) //将dev目录文件的i节点保存在
    inode_table[32]的指定表项内并将该表项指针返回

        return NULL;

}

```

}

inode_table[32]用来管理所有被打开文件的i节点；iget（）函数根据i节点号和设备号，将文件i节点载入inode_table[32]。

获得dev目录文件i节点的情景如图4-4所示。

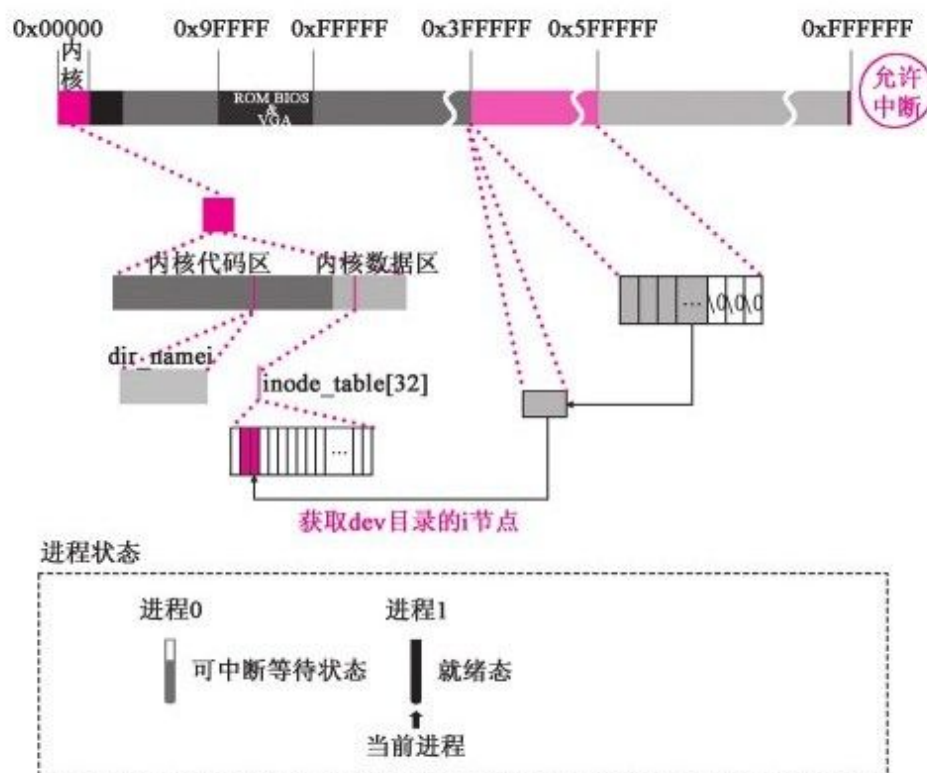


图 4-4 获得dev的i节点

4.确定dev目录文件i节点为枝梢（topmost）i
节点

获取枝梢i节点、目标文件i节点的执行路线如图4-5所示。

图 4-5 获取枝梢i节点、目标文件i节点的执行 路线图

目录项、目录文件以及文件i节点的关系图，
在第3章3.3.3节中已经介绍。

继续遍历并解析"/dev/tty0"这个路径名。解析
的技术路线与前面解析dev目录项一致，但结果有
所不同，具体执行代码如下：

//代码路径: fs/namei.c:

```
static struct m_inode * get_dir (const char * pathname)

{

.....

if ( (c=get_fs_byte (pathname) ) == '/') {

inode=current->root;

pathname++;
```



```

    }else if I

inode=current->pwd;

else

return NULL; /*empty name is bad*/

inode->i_count++;

while (1) { //循环以下过程，直到找到枝梢i节点为止

    thisname=pathname; //前面的解析工作使thisname指向/dev/tty0路径
    名中tty0前面的't'

    if ( ! S_ISDIR (inode->i_mode) || ! permission
    (inode,MAY_EXEC) ) {

        iput (inode) ;

        return NULL;

    }

    //继续查找'/'，最后遍历完路径字符串，c为'\0'跳出循环

    for (namelen=0; (c=get_fs_byte (pathname++)) && (c!
    ='/') ; namelen++)

        /*nothing*/;

    if ( ! c)

        return inode; //将枝梢i节点返回

```

```

if ( ! (bh=find_entry (&inode,thisname,namelen, &de) ) ) {

    iput (inode) ;

    return NULL;

}

inr=de->inode;

idev=inode->i_dev;

brelse (bh) ;

    iput (inode) ;

    if ( ! (inode=iget (idev,inr) ) )

        return NULL;

}

}

```

C语言规定，字符串的特征是以'\0'结尾，遍历到'\0'，`c=get_fs_byte (pathname++)` 这个条件为假，跳出循环。这意味着遍历时最近一次检测到的'\0'，是此字符串中最后一个'\0'；它后面的tty0就

是目标文件的文件名，这个文件名存储在'/'前面dev目录文件中的tty0目录项内；通过dev目录文件，就可以最终找到tty0文件。我们将dev目录文件的i节点，命名为枝梢i节点。

获取枝梢i节点后，还需要确定目标文件目录名的“首字符地址”和“名字长度”这两个信息，用它们与虚拟盘中存储的目录名进行比对。这样，围绕枝梢i节点开展的工作就完成了。获取目录名信息的代码如下：

//代码路径： fs/namei.c:

```
static struct m_inode * dir_namei (const char * pathname,
int * namelen,const char ** name)
{
char c;

const char * basename;
```

```

struct m_inode * dir;

if ( ! (dir=get_dir (pathname) ) ) //获取i节点的执行函数

return NULL;

basename=pathname;

//逐个遍历/dev/tty0字符串，每次循环都将一个字符复制给c，直到
字符串结束

while (c=get_fs_byte (pathname++))

if (c=='/')

basename=pathname;

*namelen=pathname-basename-1; //确定tty0名字的长度

*name=basename; //得到tty0中第一个't'字符的地址

return dir;

}

```

5.确定tty0文件的i节点

第二阶段获取目标文件i节点的代码与前面获取枝梢i节点的代码的技术路线大体一致，也是通

过调用find_entry () 函数，将目标文件的目录项 (tty0) 载入缓冲块，并从目录项中获得i节点号，再调用iget () 函数，通过i节点号和设备号，在虚拟盘上获取tty0文件的i节点，最终将此i节点返回，执行代码如下：

//代码路径： fs/namei.c:

```
int open_namei (const char * pathname,int flag,int mode,
struct m_inode ** res_inode)
{
    const char * basename;

    int inr,dev,namelen;

    struct m_inode * dir, *inode;

    struct buffer_head * bh;

    struct dir_entry * de;

    if ( (flag&O_TRUNC) && ! (flag&O_ACCMODE) )
```

```
flag|=O_WRONLY;
```

```
mode&=0777&~current->umask;
```

```
mode|=I_REGULAR;
```

```
if (! (dir=dir_namei (pathname, &namelen, &  
basename) )) //得到了枝梢i节点
```

```
return-ENOENT;
```

```
if (! namelen) {/*special case: '/usr/etc*/
```

```
if (! (flag& (O_ACCMODE|O_CREAT|O_TRUNC) )) {
```

```
*res_inode=dir;
```

```
return 0;
```

```
}
```

```
iput (dir) ;
```

```
return-EISDIR;
```

```
}
```

//通过枝梢i节点，以及掌握的关于tty0的情况，将tty0这一目录项载入缓冲块，de指向tty0目录项

```
bh=find_entry (&dir,basename,namelen, &de) ;
```

if (! bh) {//tty0目录项找到了，缓冲块不可能为空，if中此时不会执行

```

if ( ! (flag&O_CREAT) ) {

input (dir) ;

return-ENOENT;

}

if ( ! permission (dir,MAY_WRITE) ) {

input (dir) ;

return-EACCES;

}

inode=new_inode (dir->i_dev) ;

if ( ! inode) {

input (dir) ;

return-ENOSPC;

}

inode->i_uid=current->euid;

inode->i_mode=mode;

inode->i_dirt=1;

bh=add_entry (dir,basename,namelen, &de) ;

```

```

if (! bh) {

inode->i_nlinks--;

iput (inode) ;

iput (dir) ;

return-ENOSPC;

}

de->inode=inode->i_num;

bh->b_dirt=1;

brelse (bh) ;

iput (dir) ;

*res_inode=inode;

return 0;

}

inr=de->inode; //得到i节点号

dev=dir->i_dev; //得到虚拟盘的设备号

brelse (bh) ;

iput (dir) ;

```



```

if (flag & O_EXCL)

return -EEXIST;

if ( ! (inode = iget (dev, inr) ) ) //tty0 这个文件的i节点

return -EACCES;

if ( (S_ISDIR (inode->i_mode) && (flag &
O_ACCMODE) ) ||

! permission (inode, ACC_MODE (flag) ) ) {

input (inode) ;

return -EPERM;

}

inode->i_atime = CURRENT_TIME;

if (flag & O_TRUNC)

truncate (inode) ;

*res_inode = inode; //将此i节点传递给sys_open

return 0;

}

```

查找tty0文件i节点的情景如图4-6所示。

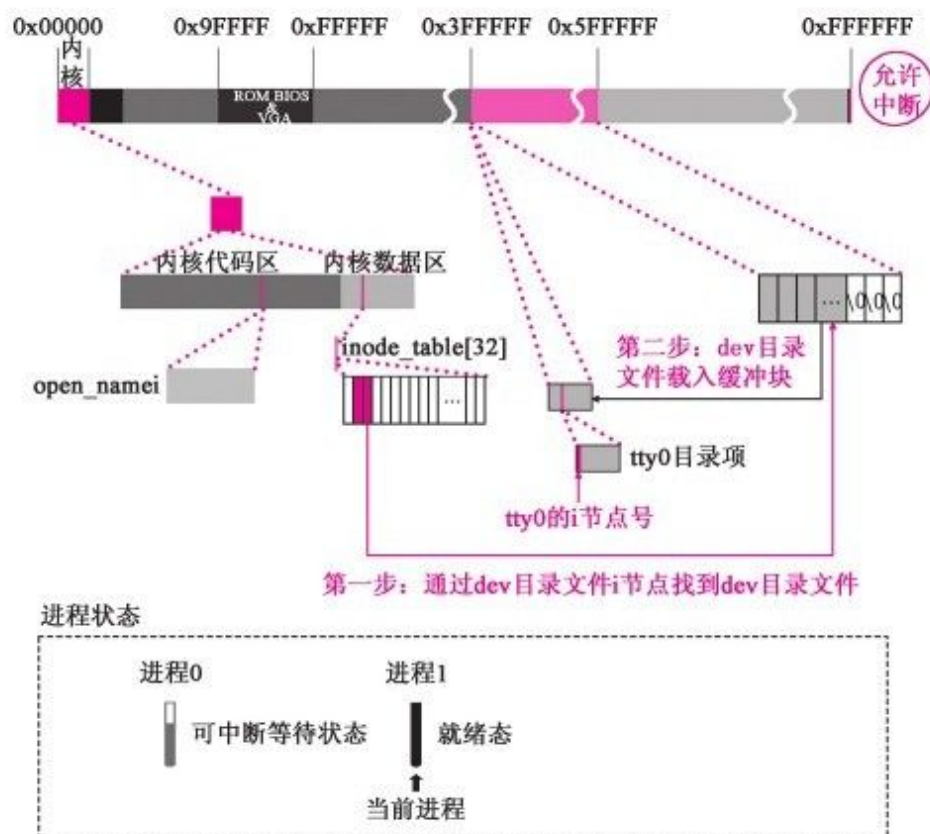


图 4-6 查找tty0文件i节点

6.确定tty0是字符设备文件

分析tty0文件的i节点属性i_mode，会得知它是设备文件，再通过i节点中的i_zone[0]，确定设

备号，并对current->tty和tty_table进行设置。执行代码如下：

```
//代码路径： fs/open.c:
```

```
int sys_open (const char * filename,int flag,int mode)
```

```
{
```

```
.....
```

```
if ( (i=open_namei (filename,flag,mode, &inode) ) < 0) {
```

```
current->filp[fd]=NULL;
```

```
f->f_count=0;
```

```
return i;
```

```
}
```

```
/*ttys are somewhat special (ttyxx major==4, tty major==5) */
```

```
if (S_ISCHR (inode->i_mode) ) //通过检测tty0文件的i节点属性，得知它是设备文件
```

```
if (MAJOR (inode->i_zone[0]) ==4) { //得知设备号是4
```

```
if (current->leader&&current->tty<0) {
```

```

//设置当前进程的tty号为该i节点的子设备号

current->tty=MINOR (inode->i_zone[0]) ;

//设置当前进程tty对应的tty表项的父进程组号为进程的父进程组号

tty_table[current->tty].pgrp=current->pgrp;

}

}else if (MAJOR (inode->i_zone[0]) ==5)

if (current->tty<0) {

input (inode) ;

current->filp[fd]=NULL;

f->f_count=0;

return-EPERM;

}

/*Likewise with block-devices: check for floppy_change*/

if (S_ISBLK (inode->i_mode) )

check_disk_change (inode->i_zone[0]) ;

f->f_mode=inode->i_mode;

f->f_flags=flag;

```

```

f->f_count=1;

f->f_inode=inode;

f->f_pos=0;

return (fd) ;

}

```

对i节点属性进行分析和相关设置的情景如图4-7所示。

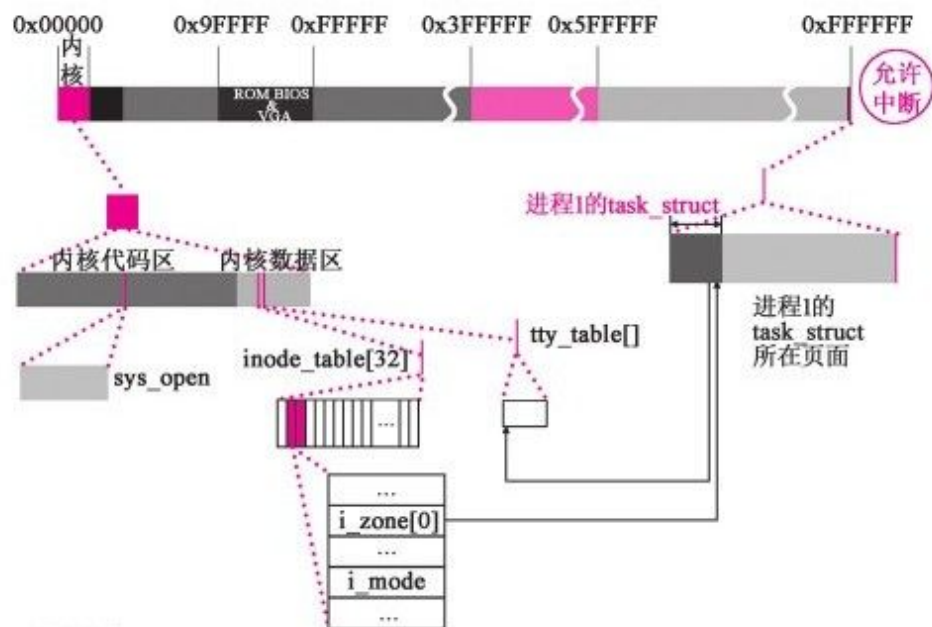


图 4-7 处理当前进程的tty



图 4-7 (续)

7. 设置file_table[0]

`sys_open` () 最后要针对`file_table[64]`中与进程1的`filp[20]`对应的表项`file_table[0]`进行设置。这样，系统通过`file_table[64]`，建立了进程1与`tty0`文件（标准输入设备文件）i节点的对应关系。执行代码如下：

//代码路径: fs/open.c:

```
int sys_open (const char * filename,int flag,int mode)
{
.....
```

```

if ( (i=open_namei (filename,flag,mode, &inode) ) < 0) {

current->filp[fd]=NULL;

f->f_count=0;

return i;

}

/*ttys are somewhat special (ttyxx major==4, tty major==5) */

if (S_ISCHR (inode->i_mode) )

if (MAJOR (inode->i_zone[0]) ==4) {

if (current->leader&&current->tty<0) {

current->tty=MINOR (inode->i_zone[0]) ;

tty _table[current->tty].pgrp=current->pgrp;

}

} else if (MAJOR (inode->i_zone[0]) ==5)

if (current->tty<0) {

input (inode) ;

current->filp[fd]=NULL;

f->f_count=0;

```

```
return-EPERM;

}

/*Likewise with block-devices: check for floppy_change*/

if (S_ISBLK (inode->i_mode) )

check_disk_change (inode->i_zone[0]) ;

f->f_mode=inode->i_mode; //用该i节点属性，设置文件属性

f->f_flags=flag; //用flag参数，设置文件标识

f->f_count=1; //将文件引用计数加1

f->f_inode=inode; //文件与i节点建立关系

f->f_pos=0; //将文件读写指针设置为0

return (fd) ;

}
```

设置file_table[0]及返回文件句柄的情景如图4-8所示。

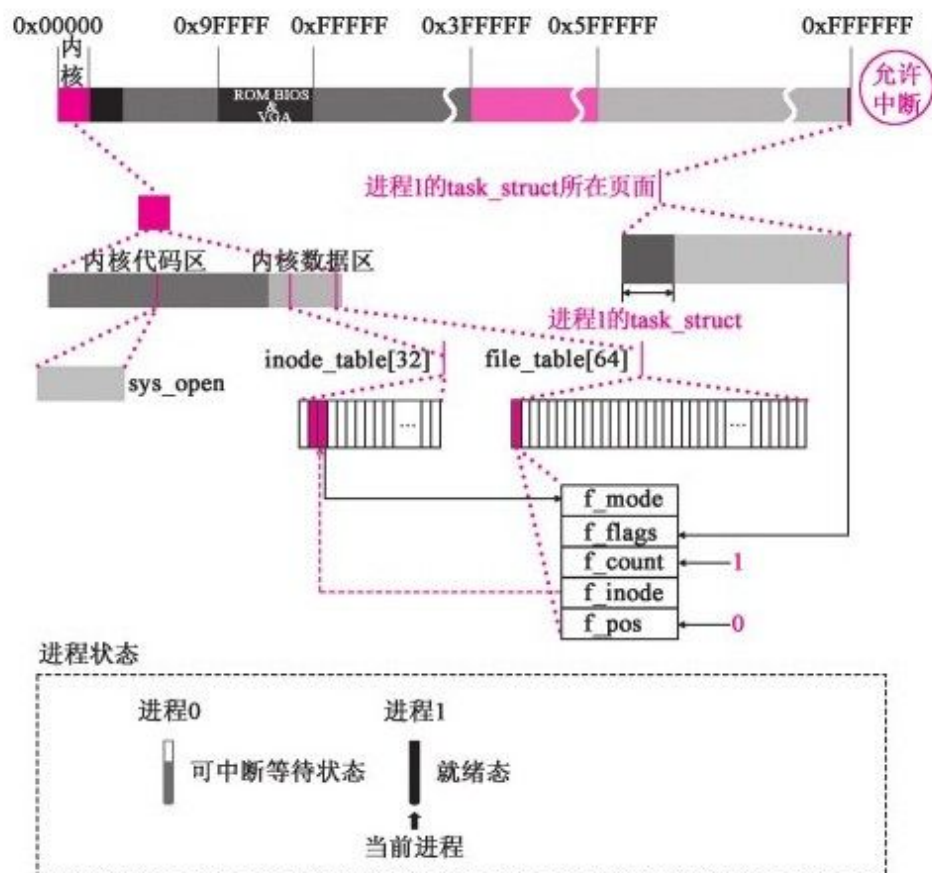


图 4-8 设置file_table[0]及返回文件句柄

4.1.2 打开标准输出、标准错误输出设备文件

4.1.1 节讲解了通过`open ()` 函数打开标准输入设备文件。下面要打开标准输出、标准错误输出设备文件，不同之处在于这里用的是复制文件句柄的方法。

`open ()` 函数返回后，进程1在`tty0`文件已经被打开的基础上，通过调用`dup ()` 函数，复制文件句柄，一共复制了两次。

第一次执行代码如下：

```
//代码路径: init/main.c:
```

```
void init (void)
```

```

{

int pid,i;

setup ( (void *) &drive_info) ;

(void) open ("/dev/tty0", O_RDWR, 0) ;

(void) dup (0) ; //复制句柄，构建标准输出设备

(void) dup (0) ;

printf ("%d buffers=%d bytes buffer space\n\r", NR_BUFFERS,
NR_BUFFERS * BLOCK_SIZE) ;

printf ("Free mem: %d bytes\n\r", memory_end-
main_memory_start) ;

if (! (pid=fork () ) ) { //if下为进程2的代码

close (0) ;

if (open ("/etc/rc", O_RDONLY, 0) )

_exit (1) ;

execve ("/bin/sh", argv_rc,envp_rc) ;

_exit (2) ;

}

```

```
if (pid > 0)

while (pid != wait (&i) )

/*nothing*/;

.....

}
```

`dup ()` 函数最终会映射到 `sys_dup ()` 这个系统调用函数中（这一映射过程与 `open ()` 函数到 `sys_open ()` 函数的映射过程大体一致），并调用到 `dupfd ()` 函数中，复制文件句柄，执行代码如下：

//代码路径：fs/fcntl.c:

```
int sys_dup (unsigned int fildes) //dup对应的系统调用函数

{

return dupfd (fildes, 0) ; //执行复制句柄

}
```

确定具备复制条件后，在进程1的filp[20]中寻找到空闲项，此时会找到第二项。即filp[1]。将filp[0]中存储的tty0文件指针复制进filp[1]中，并将file_table[0]中f_count文件引用计数这一字段的数值累加为2，以此获得进程1打开标准输出设备文件tty0的效果。执行代码如下：

```
//代码路径： fs/fcntl.c:

static int dupfd (unsigned int fd,unsigned int arg)

{

    if (fd >= NR_OPEN || ! current-> filp[fd]) //检测是否具备复制文件句柄的条件

        return -EBADF;

    if (arg >= NR_OPEN)

        return -EINVAL;

    while (arg < NR_OPEN)
```

```
    if (current->filp[arg]) //在进程1的filp[20]中寻找空闲项（此时是第二项），以便复制
```

```
        arg++;
```

```
    else
```

```
        break;
```

```
    if (arg >= NR_OPEN)
```

```
        return -EMFILE;
```

```
    current->close_on_exec &= ~ (1 < < arg) ;
```

```
    //复制文件句柄，建立标准输出设备，并相应增加文件引用计数，f_count为2
```

```
    (current->filp[arg]=current->filp[fd]) -> f_count++;
```

```
    return arg;
```

```
}
```

打开标准输出设备文件的情景如图4-9所示。

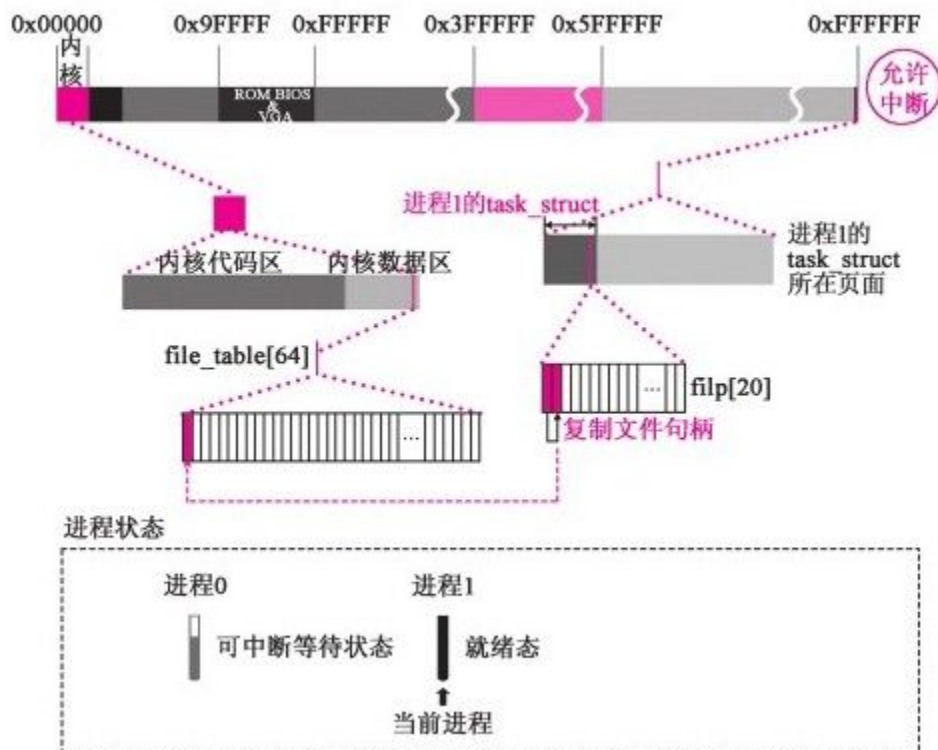


图 4-9 复制文件句柄，打开标准输出设备文件

dup返回后，进程1再次调用dup（）函数，第二次复制文件句柄，构建标准错误输出设备。

执行代码如下：

//代码路径：init/main.c:

```
void init (void)
```

```

{

int pid,i;

setup ( (void *) &drive_info) ;

(void) open ("/dev/tty0", O_RDWR, 0) ;

(void) dup (0) ; //复制句柄，构建标准输出设备

(void) dup (0) ; //继续复制句柄，构建标准错误输出设备

printf ("%d buffers=%d bytes buffer space\n\r", NR_BUFFERS,
NR_BUFFERS * BLOCK_SIZE) ;

printf ("Free mem: %d bytes\n\r", memory_end-
main_memory_start) ;

if (! (pid=fork () ) ) { //if下为进程2的代码

close (0) ;

if (open ("/etc/rc", O_RDONLY, 0) )

_exit (1) ;

execve ("/bin/sh", argv_rc,envp_rc) ;

_exit (2) ;

}

```



```
if (pid > 0)

while (pid != wait (&i) )

/*nothing*/;

.....

}
```

再次执行到dupfd () 函数中，与前面的技术路线完全一致，内核还会在进程1的filp[20]中寻找空闲项，但这次找到的是第三项，即filp[2]。与前面相同，将filp[0]中存储的tty0文件指针复制进filp[2]中，并将file_table[0]中f_count文件引用计数这一字段的数值累加为3，以此获得进程1已经打开标准错误输出设备文件的相同效果。

执行代码如下：

```
//代码路径： fs/fcntl.c:
```

```

static int dupfd (unsigned int fd,unsigned int arg)

{

    if (fd>=NR_OPEN||! current->filp[fd]) //检测是否具备复制文件
句柄的条件

    return-EBADF;

    if (arg>=NR_OPEN)

    return-EINVAL;

    while (arg<NR_OPEN)

    if (current->filp[arg]) //在进程1的filp中寻找空闲项（此时是第三
项），以便复制

    arg++;

    else

    break;

    if (arg>=NR_OPEN)

    return-EMFILE;

    current->close_on_exec&=~ (1<<arg) ;

    //复制文件句柄，建立标准错误输出设备，继续增加文件引用计
数，f_count为3

    (current->filp[arg]=current->filp[fd]) -> f_count++;

```

```
return arg;

}
```

打开标准错误输出设备文件的情景如图4-10所示。

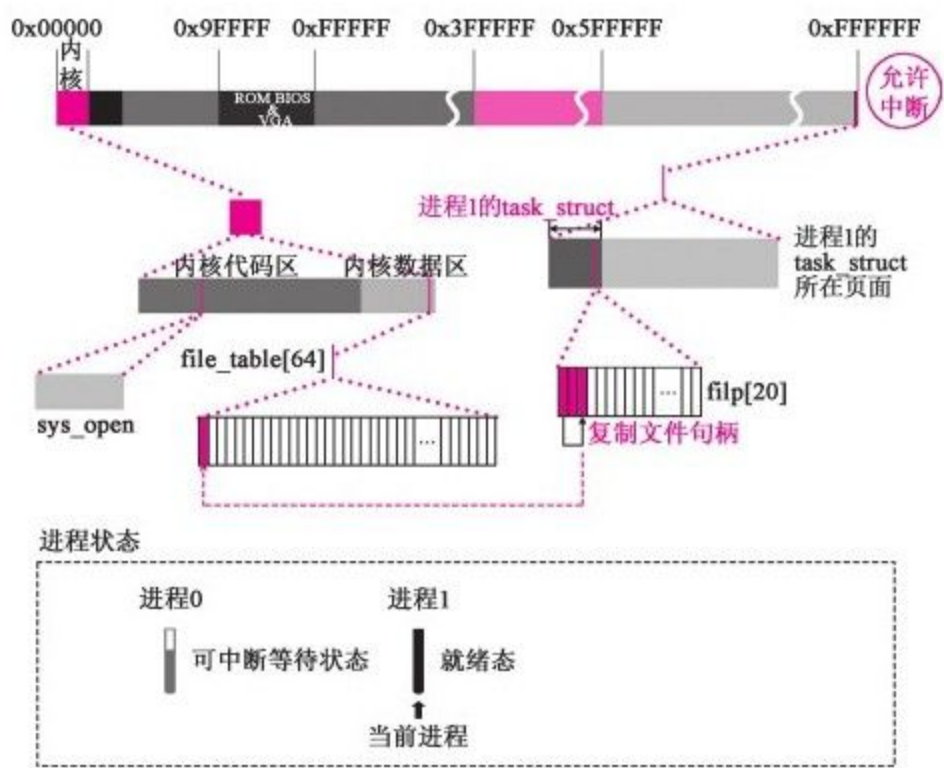


图 4-10 再次复制文件句柄，打开标准错误输出设备文件

至此，创建shell所需要的终端标准输入设备文件、标准输出设备文件和标准错误输出设备文件都已经打开，这也意味着此后可以在程序中使用printf（）函数。（stdio.h中的stdio就是standard input/output的意思）

4.2 进程1创建进程2并切换到进程2 执行

接下来，进程1将调用fork（）函数，创建进程2。

执行代码如下：

```
//代码路径： init/main.c:

void init (void)

{

int pid,i;

setup ( (void *) &drive_info) ;

(void) open ("/dev/tty0", O_RDWR, 0) ;

(void) dup (0) ;

(void) dup (0) ;
```

```
printf ("%d buffers=%d bytes buffer space\n\r", NR_BUFFERS,
NR_BUFFERS * BLOCK_SIZE) ;

printf ("Free mem: %d bytes\n\r", memory_end-
main_memory_start) ;

if ( ! (pid=fork ( ) ) ) { //进程1创建进程2

close (0) ;

if (open ("/etc/rc", O_RDONLY, 0) )

_exit (1) ;

execve ("/bin/sh", argv_rc,envp_rc) ;

_exit (2) ;

}

if (pid>0)

while (pid!=wait (&i) )

/*nothing*/;

.....

}
```

fork映射到sys_fork的过程，本书3.1.1节中已经介绍。这里的执行过程是一致的，即调用find_empty_process（）函数，为进程2寻找空闲的task，之后调用copy_process（）函数，复制进程。

执行代码如下：

```
//代码路径： kernel/system_call.s:
```

```
.....
```

```
.align 2
```

```
_sys_execve:
```

```
lea EIP（%esp）, %eax
```

```
pushl %eax
```

```
call _do_execve
```

```
addl $4, %esp
```

```
ret
```

```
.align 2
```

```
_sys_fork:
```

```
call _find_empty_process//为进程2寻找空闲task，并确定新的进程  
号
```

```
testl %eax, %eax
```

```
js 1f
```

```
push %gs
```

```
pushl %esi
```

```
pushl %edi
```

```
pushl %ebp
```

```
pushl %eax
```

```
call _copy_process//复制进程2
```

```
addl $20, %esp
```

```
1: ret
```

```
_hd_interrupt:
```

```
pushl %eax
```

```
pushl %ecx
```



```
pushl %edx

push %ds

push %es

push %fs

movl $0x10, %eax

mov %ax, %ds

mov %ax, %es

movl $0x17, %eax

mov %ax, %fs

movb $0x20, %al

outb %al, $0xA0#EOI to interrupt controller#1

jmp 1f#give port chance to breathe

.....
```

寻找新task的情景如图4-11所示:

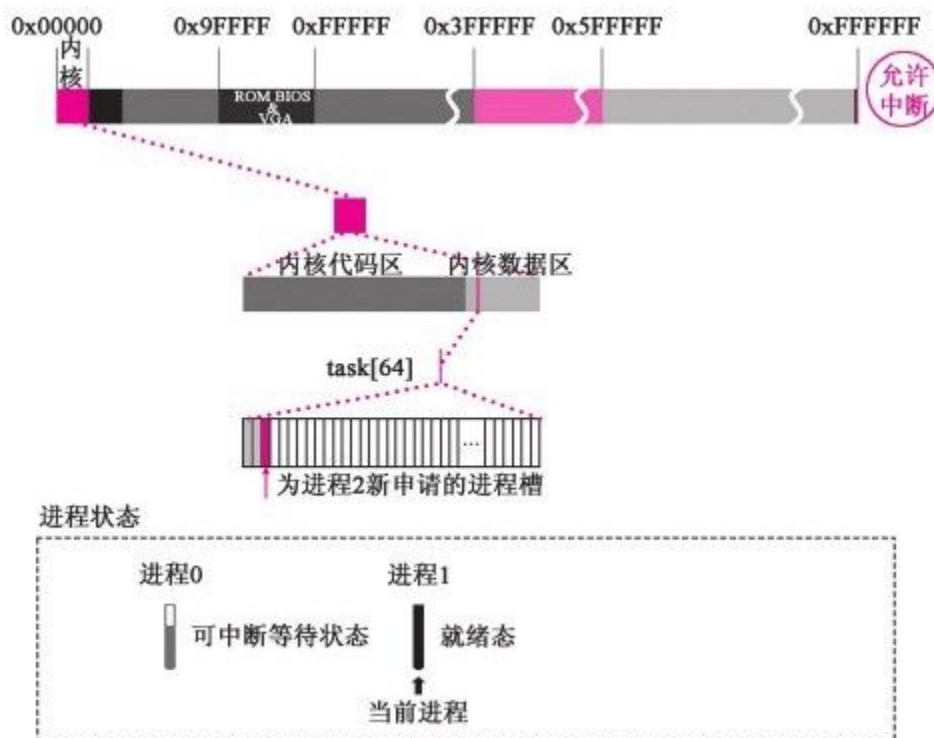


图 4-11 寻找新task

进入copy_process () 函数后，会为进程2的task_struct以及内核栈申请页面，并复制task_struct，随后对进程2的task_struct进行各种个性化设置，包括各个寄存器的设置、内存页面的管理设置、共享文件的设置、GDT表项的设置

等。其设置过程与本书第3章3.1节中进程0创建进程1的过程大体一致，执行代码如下：

//代码路径: kernel/system_call.s:

```
int copy_process (int nr,long ebp,long edi,long esi,long gs,long
none,

long ebx,long ecx,long edx,

long fs,long es,long ds,

long eip,long cs,long eflags,long esp,long ss)

{

struct task_struct * p;

int i;

struct file * f;

p= (struct task_struct *) get_free_page () ; //为进程2申请页面

if (! p)

return-EAGAIN;

task[nr]=p; //确定进程2 task_struct的地址指针载入task指定位置
```

`*p=*current; /*NOTE! this doesn't copy the supervisor stack*///复制
task_struct结构`

`p->state=TASK_UNINTERRUPTIBLE; //设置进程2为不可中断等
待状态`

`p->pid=last_pid; //对进程2进行个性化设置`

`p->father=current->pid;`

`p->counter=p->priority;`

`p->signal=0;`

`p->alarm=0;`

`p->leader=0; /*process leadership doesn't inherit*/`

`p->utime=p->stime=0;`

`p->cutime=p->cstime=0;`

`p->start_time=jiffies;`

`p->tss.back_link=0;`

`p->tss.esp0=PAGE_SIZE+ (long) p;`

`p->tss.ss0=0x10;`

`p->tss.eip=eip;`

`p->tss.eflags=eflags;`

```
p->tss.eax=0;

p->tss.ecx=ecx;

p->tss.edx=edx;

p->tss.ebx=ebx;

p->tss.esp=esp;

p->tss.ebp=ebp;

p->tss.esi=esi;

p->tss.edi=edi;

p->tss.es=es&0xffff;

p->tss.cs=cs&0xffff;

p->tss.ss=ss&0xffff;

p->tss.ds=ds&0xffff;

p->tss.fs=fs&0xffff;

p->tss.gs=gs&0xffff;

p->tss.ldt=_LDT (nr) ;

p->tss.trace_bitmap=0x80000000;

if (last_task_used_math==current)
```

```

__asm__ ("clts; fnsave%0": "m" (p->tss.i387) ) ;

if (copy_mem (nr,p) ) { //设置进程2的分页管理

task[nr]=NULL;

free_page ( (long) p) ;

return-EAGAIN;

}

for (i=0; i<NR_OPEN; i++) //以下为进程2共享进程1的文件

if (f=p->filp[i])

f->f_count++;

if (current->pwd)

current->pwd->i_count++;

if (current->root)

current->root->i_count++;

if (current->executable)

current->executable->i_count++;

set_tss_desc (gdt+ (nr<<1) +FIRST_TSS_ENTRY, & (p->
tss) ) ; //设置进程2在

```

```
set_ldt_desc (gdt+ (nr<<1) +FIRST_LDT_ENTRY, & (p->
ldt) ) ; //GDT中的表项
```

```
p->state=TASK_RUNNING; /*do this last,just in case*///设置进程2
为就绪态
```

```
return last_pid;
```

```
}
```

复制进程及部分个性化设置的情景如图4-12所示。

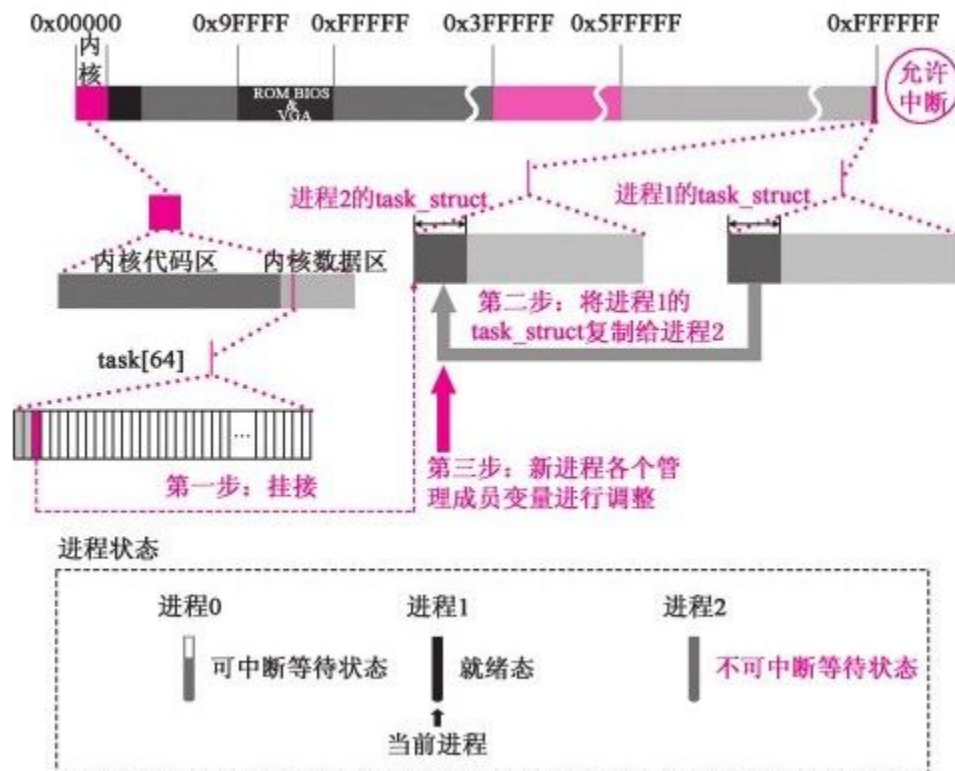


图 4-12 复制进程及部分设置

为进程2复制页表和设置页目录项的情景如图4-13所示。

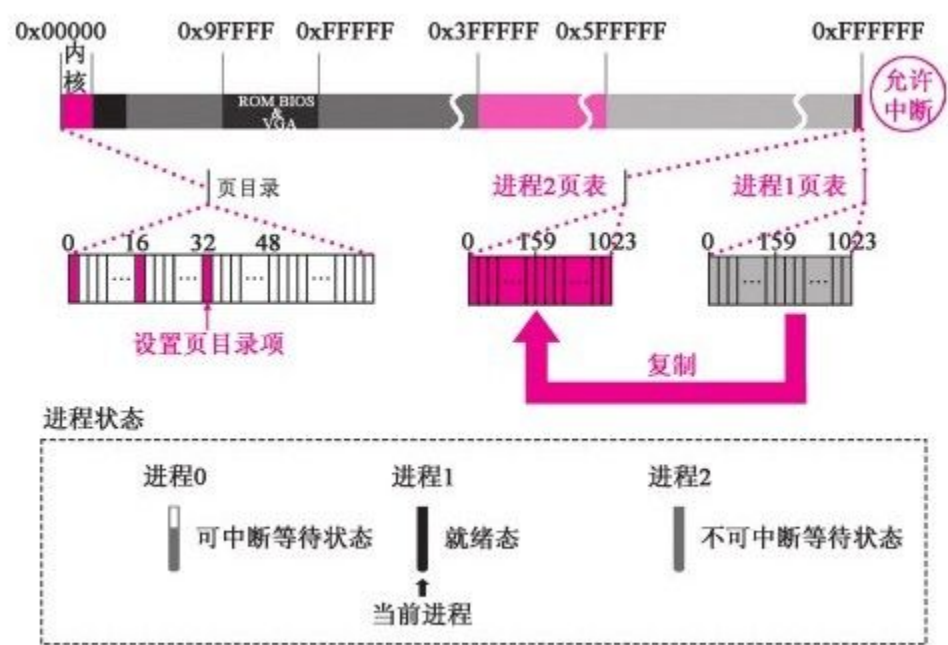


图 4-13 复制页表和设置页目录项

对进程2共享进程1的文件进行调整的情景如图4-14所示。

进程2创建完毕后，`fork ()` 函数返回，返回值为2，因此！`(pid=fork ())` 值为假（理由在本书3.1.7节中已经介绍），于是调用`wait ()` 函数。此函数的功能是：如果进程1有等待退出的子进程，就为该进程的退出做善后工作；如果有子进程，但并不等待退出，则进行进程切换；如果没子进程，函数返回。

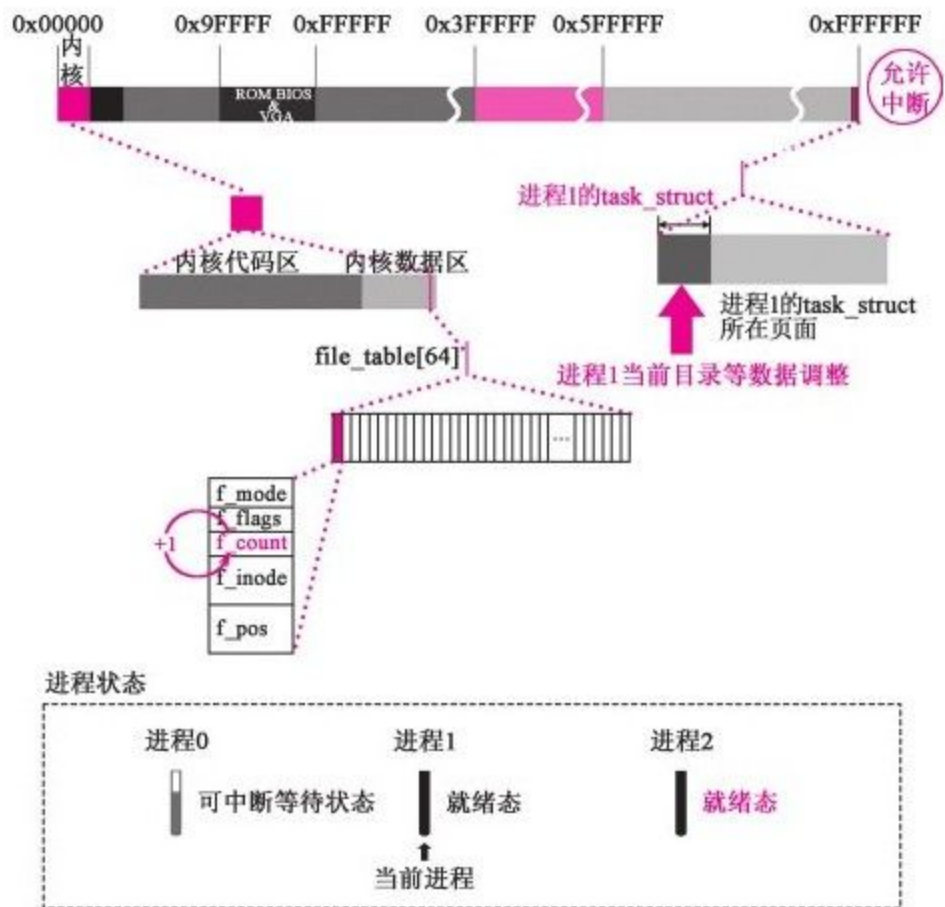


图 4-14 调整进程2共享进程1的文件

执行代码如下：

//代码路径：init/main.c:

```
void init (void)
```

```
{
```

```

int pid,i;

setup ( (void *) &drive_info) ;

(void) open ("/dev/tty0", O_RDWR, 0) ;

(void) dup (0) ;

(void) dup (0) ;

printf ("%d buffers=%d bytes buffer space\n\r", NR_BUFFERS,
NR_BUFFERS * BLOCK_SIZE) ;

printf ("Free mem: %d bytes\n\r", memory_end-
main_memory_start) ;

if (! (pid=fork () ) ) { //括号里面为进程2执行的代码

close (0) ;

if (open ("/etc/rc", O_RDONLY, 0) )

_exit (1) ;

execve ("/bin/sh", argv_rc,envp_rc) ;

_exit (2) ;

}

if (pid>0)

```

```
while (pid != wait (&i) ) //进程1等待子进程退出，最终会切换到进程2执行
```

```
/*nothing*/;
```

```
.....
```

```
}
```

`wait ()` 函数最终会映射到系统调用函数 `sys_waitpid ()` 中执行，映射方式与 `fork` 到 `sys_fork` 大体一致。`sys_waitpid ()` 函数先要对所有的进程进行遍历，先确定哪个进程是进程1的子进程，由于进程1刚刚创建了子进程，即进程2，于是进程2被选中了，执行代码如下：

```
//代码路径： kernel/exit.c:
```

```
int sys_waitpid (pid_t pid,unsigned long * stat_addr,int options) //wait对应sys_waitpid系统调用
```

```
{
```

```
int flag,code;
```

```

struct task_struct ** p;

verify_area (stat_addr, 4) ;

repeat:

flag=0;

for (p=&LAST_TASK; p> &FIRST_TASK; --p) {

if (! *p||*p==current)

continue;

    if ( (*p) -> father !=current->pid) //筛选出当前进程，即进程1
的子进程，此时会是进程2

continue;

if (pid>0) {

if ( (*p) -> pid !=pid)

continue;

}else if (! pid) {

if ( (*p) -> pgrp !=current->pgrp)

continue;

}else if (pid !=-1) {

```

```

if ( (*p) -> pgrp != -pid)

continue;

}

switch ( (*p) -> state) { //判断进程2的状态

case TASK_STOPPED: //如果进程2是停止状态，将在这里处理

if ( ! (options & WUNTRACED) )

continue;

put_fs_long (0x7f, stat_addr) ;

return (*p) -> pid;

case TASK_ZOMBIE: //如果进程2是僵死状态，将在这里处理

current->cutime += (*p) -> utime;

current->cstime += (*p) -> stime;

flag = (*p) -> pid;

code = (*p) -> exit_code;

release (*p) ;

put_fs_long (code, stat_addr) ;

return flag;

```

default: //此时进程2是就绪态，所以到这里去执行，将**flag**标志设置为1并跳出循环

```
flag=1;  
  
continue;  
  
}  
  
}  
  
.....  
  
}
```

查找进程1的子进程的情景如图4-15所示。

再对进程2进行分析，确定进程2并不准备退出，于是设置**flag**标志为1，该标志将导致进程切换，执行代码如下：

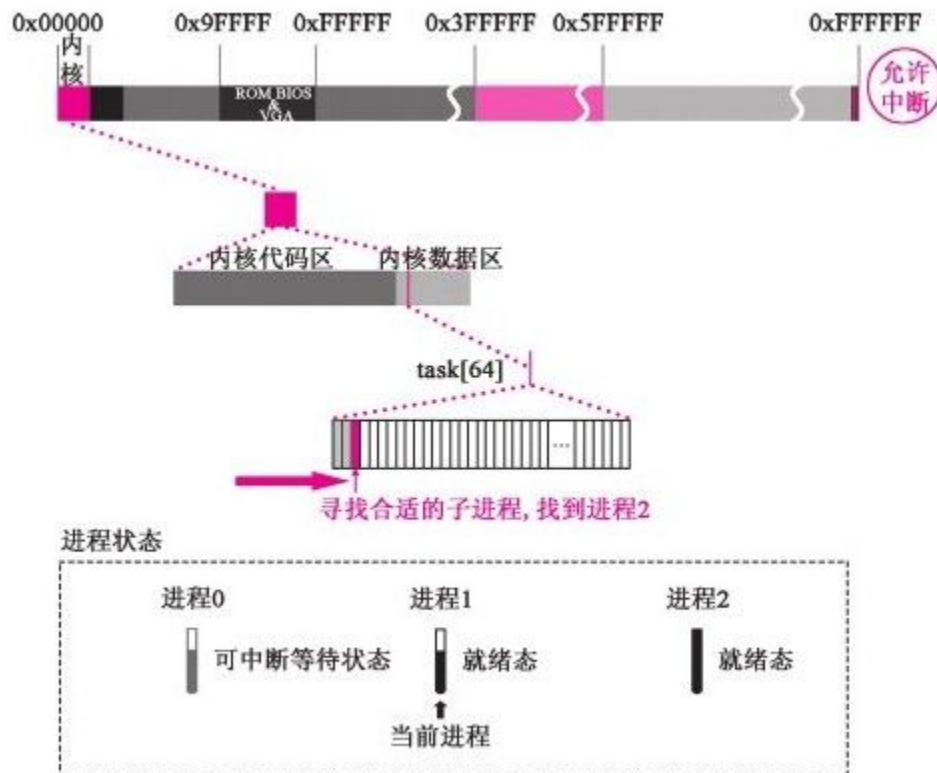


图 4-15 查找进程1的子进程

//代码路径: kernel/exit.c:

```
int sys_waitpid (pid_t pid,unsigned long * stat_addr,int
options) //wait对应sys_waitpid系统调用
```

```
{
```

```
int flag,code;
```

```
struct task_struct ** p;
```

```
verify_area (stat_addr, 4) ;
```


repeat:

flag=0;

for (p=&LAST_TASK; p> &FIRST_TASK; --p) {

if (! *p||*p==current)

continue;

if ((*p) -> father! =current->pid) //筛选出当前进程，即进程1
的子进程，此时会是进程2

continue;

if (pid>0) {

if ((*p) -> pid! =pid)

continue;

}else if (! pid) {

if ((*p) -> pgrp! =current-> pgrp)

continue;

}else if (pid! =-1) {

if ((*p) -> pgrp! =-pid)

continue;

```
}
```

```
switch ( (*p) -> state) { //判断进程2的状态
```

```
case TASK_STOPPED: //如果进程2是停止状态，将在这里处理
```

```
if ( ! (options & WUNTRACED) )
```

```
continue;
```

```
put_fs_long (0x7f, stat_addr) ;
```

```
return (*p) -> pid;
```

```
case TASK_ZOMBIE: //如果进程2是僵死状态，将在这里处理
```

```
current->cutime+= (*p) -> utime;
```

```
current->cstime+= (*p) -> stime;
```

```
flag= (*p) -> pid;
```

```
code= (*p) -> exit_code;
```

```
release (*p) ;
```

```
put_fs_long (code, stat_addr) ;
```

```
return flag;
```

default: //此时进程2是就绪态，所以到这里来执行，将flag标志设置为1并跳出循环

```

flag=1;

continue;

}

}

.....

}

```

判断进程2的状态并设置flag标志的情景如图4-16所示。

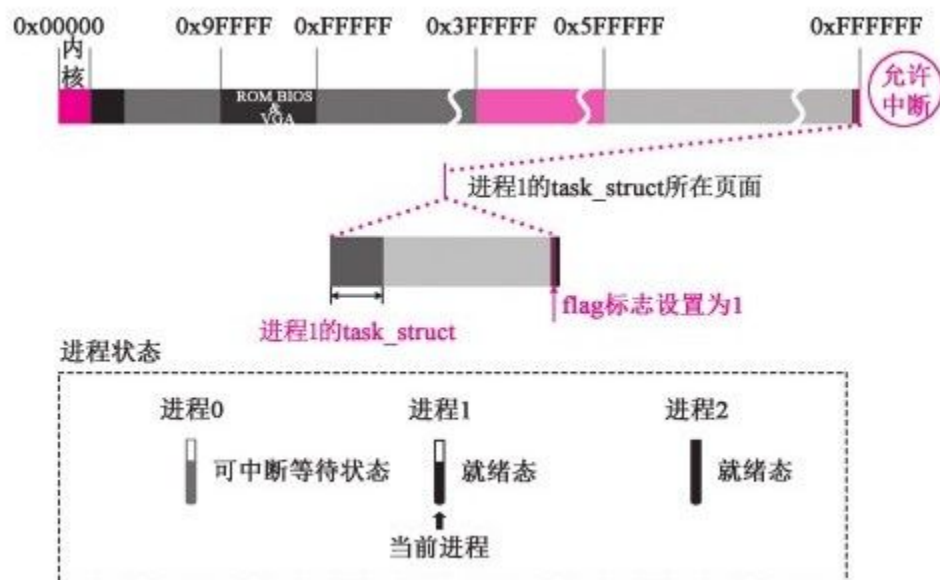


图 4-16 判断进程2的状态并设置flag标志

进入if (flag) 去执行，内核先将进程1的状态设置为可中断等待状态，之后，就调用schedule

() 函数切换进程，此时唯一处于就绪态的是进程2，因此，代码将切换到进程2去执行（schedule的执行过程已在第3章3.2节中介绍），执行代码如下：

//代码路径： kernel/exit.c:

```
int sys_waitpid (pid_t pid,unsigned long * stat_addr,int  
options) //wait对应sys_waitpid系统调用
```

```
{
```

```
.....
```

```
switch ( (*p) -> state) { //判断进程2的状态
```

```
case TASK_STOPPED: //如果进程2是停止状态，将在这里处理
```

```
if ( ! (options&WUNTRACED) )
```

```
continue;
```

```
put_fs_long (0x7f,stat_addr) ;
```

```
return (*p) -> pid;
```

```
case TASK_ZOMBIE: //如果进程2是僵死状态，将在这里处理
```

```
current->cutime+= (*p) -> utime;
```

```
current->cstime+= (*p) -> stime;
```

```
flag= (*p) -> pid;
```

```
code= (*p) -> exit_code;
```

```
release (*p) ;
```

```
put_fs_long (code,stat_addr) ;
```

```
return flag;
```

```
default: //此时进程2是就绪态，所以到这里来执行，将flag标志设置为1并跳出循环
```

```
flag=1;
```

```
continue;
```

```
}
```

```
}
```

```
if (flag) {
```

```
if (options & WNOHANG)
```

```
return 0;
```

`current->state=TASK_INTERRUPTIBLE;` //此时得知进程1没有退出的子进程，所以将其设置为可中断等待状态

```
schedule (); //切换到进程2去执行
```

```
if (! (current->signal &= ~ (1 << (SIGCHLD-1) ) ) )
```

```
goto repeat;
```

```
else
```

```
return-EINTR;
```

```
}
```

```
return-ECHILD;
```

```
}
```

切换到进程2的情景如图4-17所示。



图 4-17 切换到进程2

4.3 加载shell程序

4.3.1 关闭标准输入设备文件，打开rc文件

轮转到进程2后，进程2最开始的执行技术路线，与本书3.3节中介绍的轮转到进程1执行的技术路线大体一致。在确定if（！（pid=fork（）））这条语句中条件为真后，调用close（）函数来关闭标准输入设备文件，并用rc文件替换它，执行代码如下：

```
//代码路径：init/main.c:
```

```
void init (void)
```

```
{
```



```

int pid,i;

setup ( (void *) &drive_info) ;

(void) open ("/dev/tty0", O_RDWR, 0) ;

(void) dup (0) ;

(void) dup (0) ;

printf ("%d buffers=%d bytes buffer space\n\r", NR_BUFFERS,
NR_BUFFERS * BLOCK_SIZE) ;

printf ("Free mem: %d bytes\n\r", memory_end-
main_memory_start) ;

if (! (pid=fork () ) ) {

close (0) ; //关闭标准输入设备文件

if (open ("/etc/rc", O_RDONLY, 0) ) //用rc文件替换该设备文
件

_exit (1) ;

execve ("/bin/sh", argv_rc,envp_rc) ; //加载shell程序

_exit (2) ;

}

if (pid>0)

```

```
while (pid != wait (&i) )  
  
/*nothing*/;  
  
.....  
  
}
```

`close ()` 函数最终会映射到 `sys_close ()` 函数中执行（与前面讲解的其他以 `sys_` 为前缀的函数类似）。由于进程2继承了进程1的管理信息，因此其 `filp[20]` 中文件指针存储情况与进程1是一致的。`close (0)` 就是要将 `filp[20]` 第一项清空（就是关闭标准输入设备文件 `tty0`），并递减 `file_table[64]` 中 `f_count` 的引用计数。后面调用 `open ()` 函数，就会在 `filp[20]` 中选择第一项来建立进程2与 `rc` 文件 `i` 节点的关系，以此达到“`rc`”替换“`tty0`”的效果。`rc` 文件是脚本文件，其特点是，

文件中记录着一些命令，应用程序通过解析这些命令来确定执行任务。

`close ()` 函数的执行代码如下：

```
//代码路径： fs/open.c:

int sys_close (unsigned int fd) //close对应的系统调用函数
{

    struct file * filp;

    if (fd >= NR_OPEN)

        return -EINVAL;

    current->close_on_exec &= ~ (1 << fd) ;

    if (! (filp = current->filp[fd])) //获取进程2标准输入设备文件的指针

        return -EINVAL;

    current->filp[fd] = NULL; //进程2与该设备文件解除关系

    if (filp->f_count == 0)
```

```

panic ("Close: file count is 0") ;

if (--filp->f_count) //该设备文件引用计数递减

return (0) ;

input (filp->f_inode) ;

return (0) ;

}

```

关闭tty0文件的情景如图4-18所示。

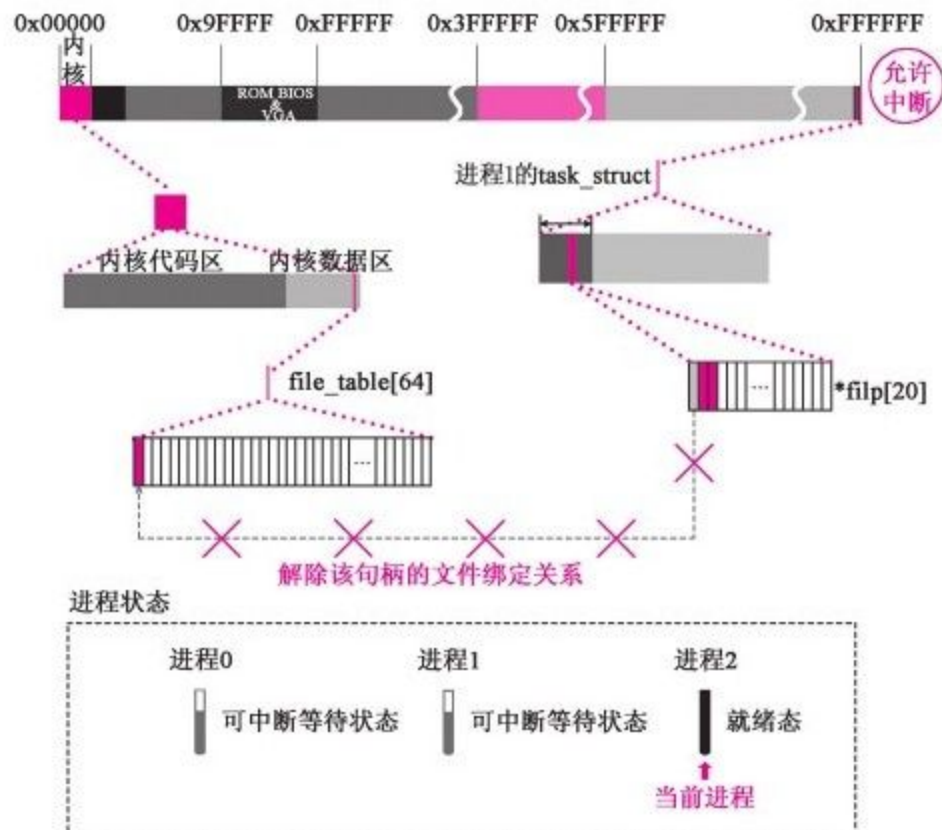


图 4-18 关闭tty0文件

打开rc文件的情景如图4-19所示。

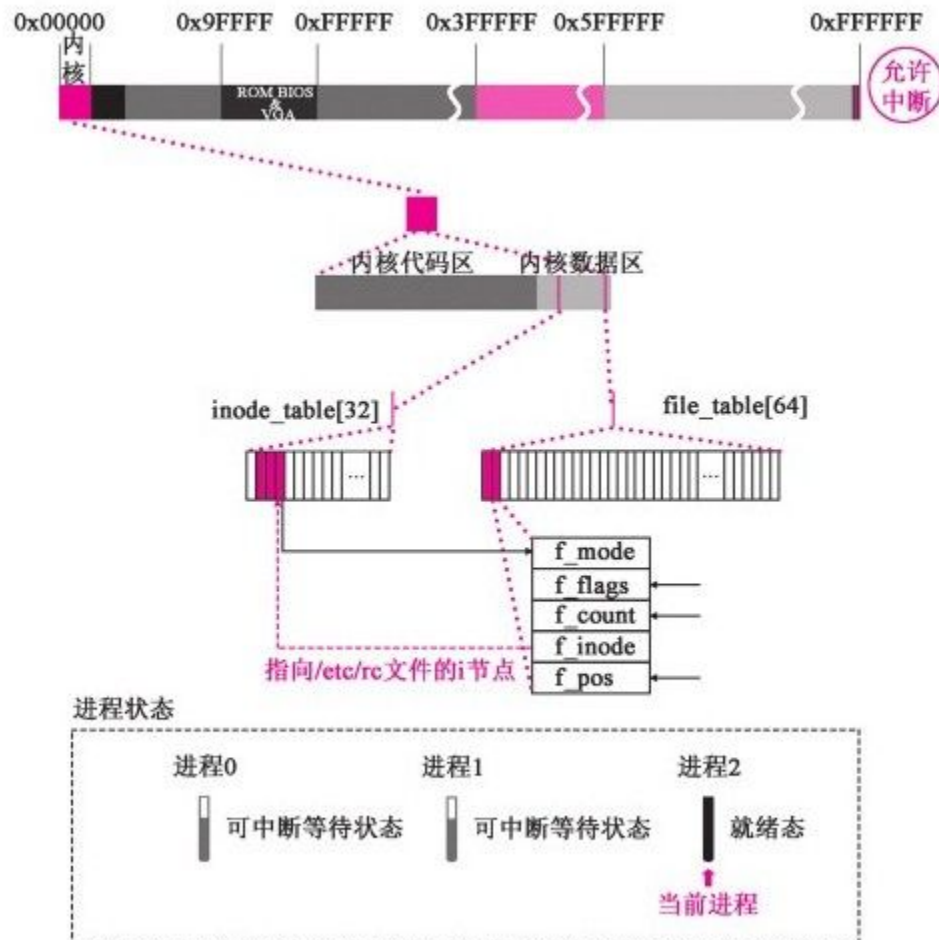


图 4-19 打开rc文件

rc文件打开后，进程2将调用execve（）函数开始加载shell程序，执行代码如下：

```
//代码路径： init/main.c:

void init (void)

{

int pid,i;

setup ( (void *) &drive_info) ;

(void) open ("/dev/tty0", O_RDWR, 0) ;

(void) dup (0) ;

(void) dup (0) ;

printf ("%d buffers=%d bytes buffer space\n\r", NR_BUFFERS,

NR_BUFFERS * BLOCK_SIZE) ;

printf ("Free mem: %d bytes\n\r", memory_end-

main_memory_start) ;

if (! (pid=fork ( ) ) ) {

close (0) ; //关闭标准输入设备文件
```

```
if (open ("/etc/rc", O_RDONLY, 0) ) //用rc文件替换该设备文件
```

```
_exit (1) ;
```

//加载shell程序，其中/bin/sh为shell文件路径，argv_rc和envp_rc分别是参数及环境变量

```
execve ("/bin/sh", argv_rc,envp_rc) ;
```

```
_exit (2) ;
```

```
}
```

```
if (pid > 0)
```

```
while (pid != wait (&i) )
```

```
/*nothing*/;
```

```
}
```

值得注意的是，参数和环境变量都已在内核代码中事先准备。具体代码如下：

```
//代码路径：init/main.c:
```

```
.....
```

static char * argv_rc[]={"/bin/sh", NULL}; //为shell进程准备的参数

static char * envp_rc[]={"HOME=/", NULL,NULL}; //为shell进程准备的环境变量

.....

execve () 函数最终会映射到sys_execve () 中去执行，代码如下：

//代码路径： kernel/system_call.s:

.....

.align 2

_sys_execve: //execve函数对应的系统调用

lea EIP (%esp) , %eax

pushl %eax//把EIP值“所在栈空间的地址值”压栈

call _do_execve//do_execve就是支持加载shell程序的主体函数

addl \$4, %esp

ret

.....

4.3.2 检测shell文件

1.检测i节点属性

`do_execve`（）开始执行后，先调用`namei`（）函数获取shell文件的i节点。此函数获取i节点的过程与4.1.1节中介绍的i节点获取过程大体一致。之后，检测i节点属性，以此确定shell程序是否具备加载条件。具体代码如下：

```
int do_execve (unsigned long * eip,long tmp,char * filename,
char ** argv,char ** envp)
{
    struct m_inode * inode;

    struct buffer_head * bh;

    struct exec ex;
```

```

unsigned long page[MAX_ARG_PAGES];

int i,argc,envc;

int e_uid,e_gid;

int retval;

int sh_bang=0;

unsigned long p=PAGE_SIZE * MAX_ARG_PAGES-4;

    if ( (0xffff&eip[1]) !=0x000f) //通过检测特权级，来判断是否
是内核调用了do_execve函数

    panic ("execve called from supervisor mode") ; //如果是，就会死
机，显然现在不是

    for (i=0; i<MAX_ARG_PAGES; i++) /*clear page-table*/

    page[i]=0; //将参数和环境变量的页面指针管理表清零

    if ( ! (inode=namei (filename) ) ) /*get executables inode*///获
取shell程序所在文件的i节点

    return-ENOENT;

    argc=count (argv) ; //统计参数个数

    envc=count (envp) ; //统计环境变量个数

restart _interp:

    if ( ! S_ISREG (inode->i_mode) ) { /*must be regular file*/

```

```

    retval=-EACCES;

    goto exec_error2;

}

i=inode->i_mode; //通过检测i节点的uid和gid属性，来判断进程2

    e_uid= (i&S_ISUID) ?inode->i_uid: current->euid; //是否有权
    限执行shell程序

    e_gid= (i&S_ISGID) ?inode->i_gid: current->egid;

    if (current->euid==inode->i_uid) //通过分析文件与当前进程的
    从属关系，调整i节点属性中的权限位

        i>>=6;

    else if (current->egid==inode->i_gid)

        i>>=3;

    if (! (i&1) && //如果用户没有权限执行该程序，则退出shell
    的加载工作

        ! ( (inode->i_mode&0111) &&suser ( ) ) ) {

        retval=-ENOEXEC;

        goto exec_error2;

    }

    .....

```

获取i节点信息的情景如图4-20所示。

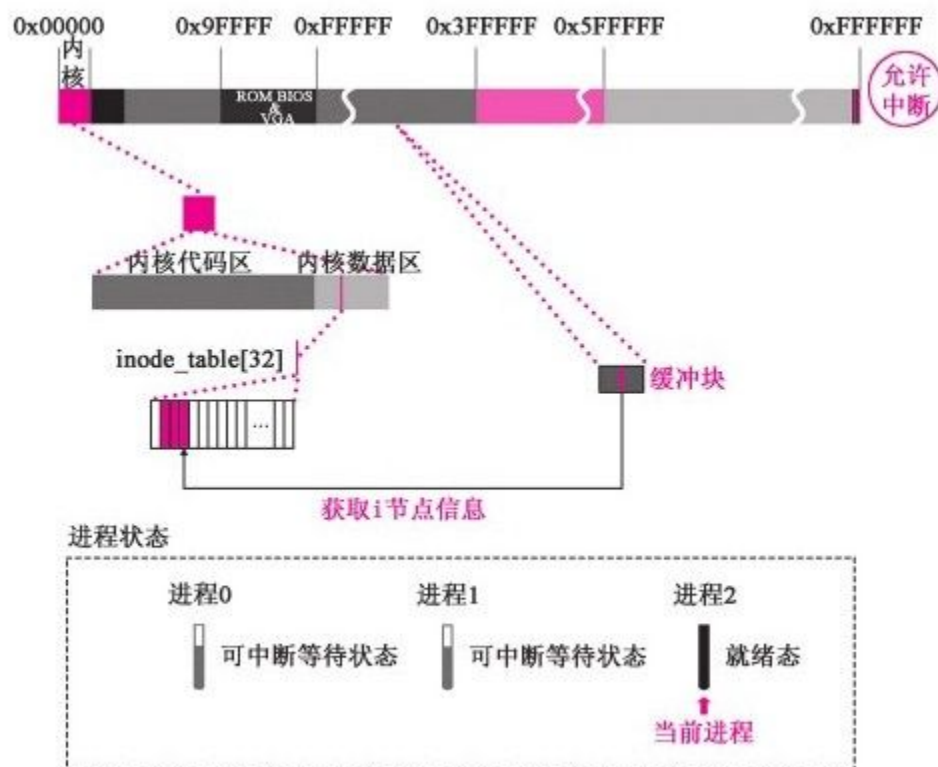


图 4-20 获取i节点信息

检测i节点属性的情景如图4-21所示。

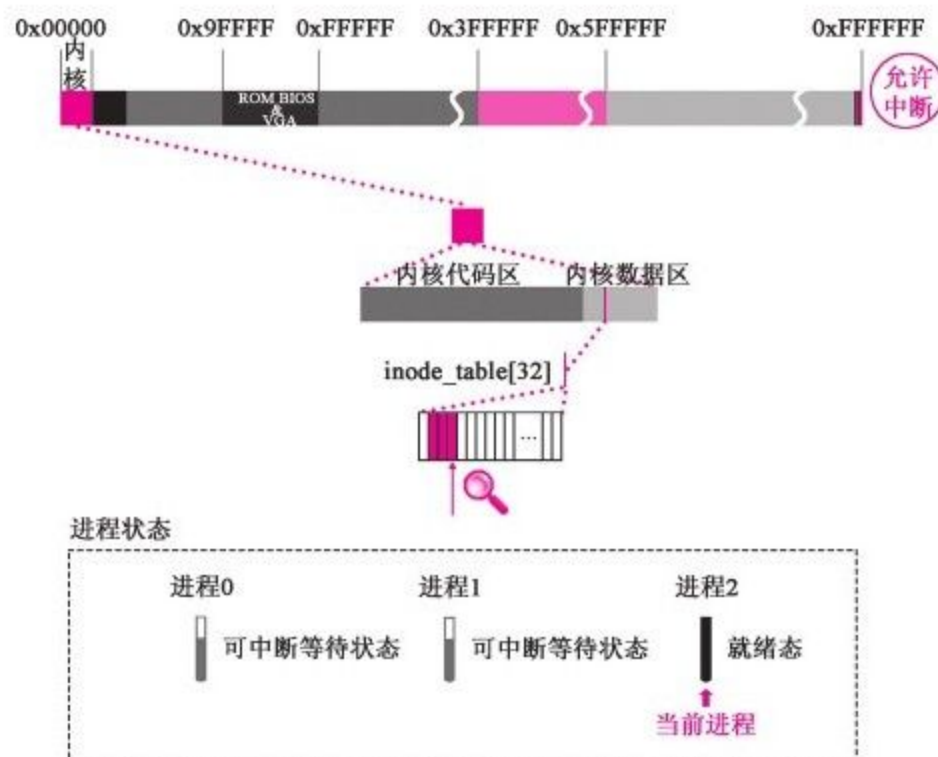


图 4-21 检测i节点属性

经检测shell文件i节点的属性得知，进程2具备执行该文件中程序的条件。

2.检测文件头属性

通过i节点中提供的设备号和块号（文件头的块号为i_zone[0]）信息，将文件头载入缓冲块并

获取其信息，如图4-22和图4-23所示。执行代码如下：

```
//代码路径: fs/exec.c:
```

```
int do_execve (unsigned long * eip,long tmp,char * filename,
```

```
char ** argv,char ** envp)
```

```
{
```

```
.....
```

```
if (! (i&1) &&
```

```
//如果用户没有权限执行该程序，则退出shell的加载工作
```

```
! ( (inode->i_mode&0111) &&suser ( ) ) ) {
```

```
retval=-ENOEXEC;
```

```
goto exec_error2;
```

```
}
```

```
    //通过i节点，确定shell文件所在设备的设备号及其文件头的块号  
    (i_zone[0])，获取文件头
```

```
if (! (bh=bread (inode->i_dev,inode->i_zone[0]) ) ) {
```

```
retval=-EACCES;
```

```
goto exec_error2;
```

```
}
```

ex=* ((struct exec *) bh->b_data) ; 从缓冲块中得到文件头的信息

```
if ( (bh->b_data[0]=='#') && (bh->b_data[1]=='! ') && (!  
sh_bang) ) {
```

```
.....
```

```
brelse (bh) ;
```

```
if (N_MAGIC (ex) !=ZMAGIC||ex.a_trsize||ex.a_drsz||
```

```
ex.a_text+ex.a_data+ex.a_bss>0x3000000||
```

```
inode->i_size<ex.a_text+ex.a_data+ex.a_syms+N_TXTOFF  
(ex) ) {
```

```
retval=-ENOEXEC;
```

```
goto exec_error2;
```

```
}
```

```
if (N_TXTOFF (ex) !=BLOCK_SIZE) {
```

```
printk ("%s: N_TXTOFF !=BLOCK_SIZE.See a.out.h.",  
filename) ;
```



```
retval=-ENOEXEC;

goto exec_error2;

}

if (! sh_bang) {

p=copy_strings (envc,envp,page,p, 0) ;

p=copy_strings (argc,argv,page,p, 0) ;

if (! p) {

retval=-ENOMEM;

goto exec_error2;

}

}

}

.....
```

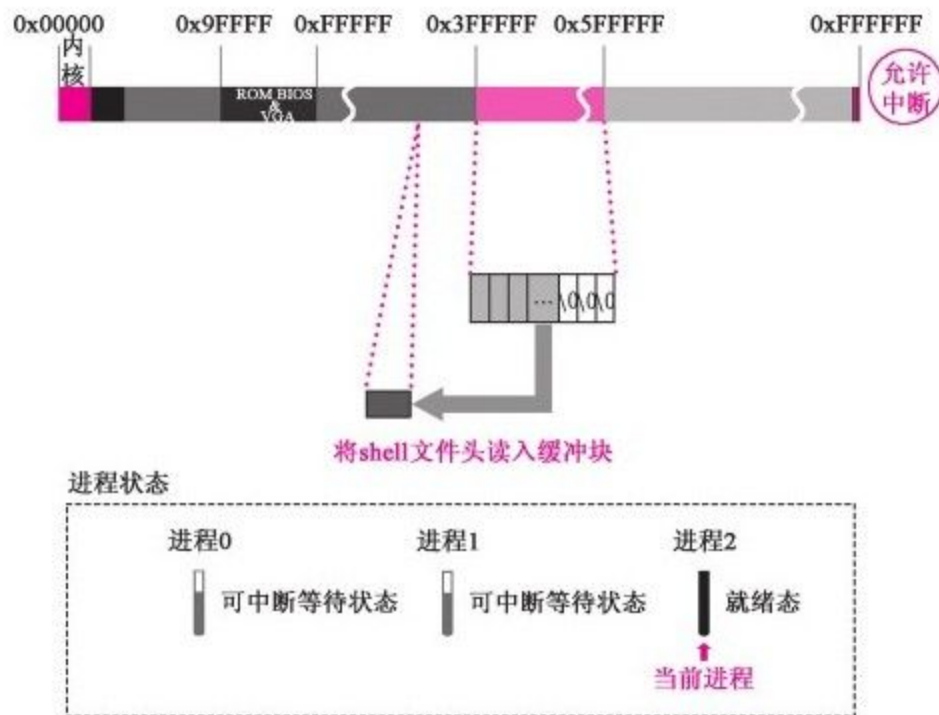


图 4-22 将shell文件头读入缓冲块

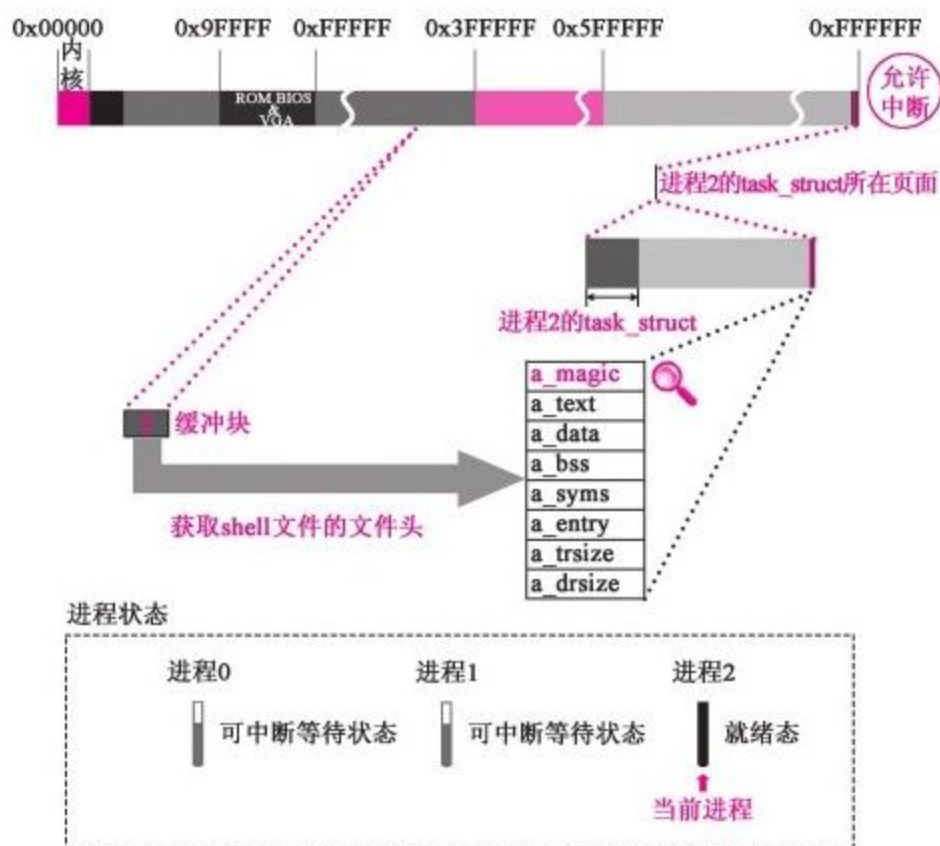


图 4-23 获取文件头

对文件头的信息进行检测，以此进一步确定shell文件的内容是否符合载入的规定。代码如下：

//代码路径：fs/exec.c:

```
int do_execve (unsigned long * eip, long tmp, char * filename,
```

```
char ** argv,char ** envp)
```

```
{
```

```
.....
```

if (! (i&1) && //如果用户没有权限执行该程序，则退出shell
的加载工作

```
! ( (inode->i_mode&0111) &&suser ( ) ) ) {
```

```
retval=-ENOEXEC;
```

```
goto exec_error2;
```

```
}
```

//通过i节点，确定shell文件所在设备的设备号及其文件头的块号
(i_zone[0])，获取文件头

```
if (! (bh=bread (inode->i_dev,inode->i_zone[0]) ) ) {
```

```
retval=-EACCES;
```

```
goto exec_error2;
```

```
}
```

ex=* ((struct exec *) bh->b_data) ; 从缓冲块中得到文件头的
信息

//检测文件头，得知shell文件并非脚本文件，所以if里面的内容不
执行

```
    if ( (bh->b_data[0]=='#') && (bh->b_data[1]=='! ') && (!
sh_bang) ) {
```

```
        .....
```

```
    brelease (bh) ;
```

```
//通过文件头中的信息，检测shell文件的内容是否符合执行规定
```

```
if (N_MAGIC (ex) !=ZMAGIC||ex.a_trsize||ex.a_drsz||
```

```
ex.a_text+ex.a_data+ex.a_bss>0x3000000||
```

```
inode->i_size<ex.a_text+ex.a_data+ex.a_syms+N_TXTOFF
(ex) ) {
```

```
    retval=-ENOEXEC;
```

```
    goto exec_error2;
```

```
}
```

```
//如果文件头大小不等于1024B，程序也不能执行
```

```
if (N_TXTOFF (ex) !=BLOCK_SIZE) {
```

```
    printk ("%s: N_TXTOFF !=BLOCK_SIZE.See a.out.h.",
filename) ;
```

```
    retval=-ENOEXEC;
```

```
    goto exec_error2;
```

```
}
```

```
if (! sh_bang) {  
  
p=copy_strings (envc,envp,page,p, 0) ;  
  
p=copy_strings (argc,argv,page,p, 0) ;  
  
if (! p) {  
  
retval=-ENOMEM;  
  
goto exec_error2;  
  
}  
  
}  
  
}  
  
.....
```

检测文件头属性的情景如图4-24所示。

经检测shell文件的文件头属性得知， shell文件中的程序具备执行条件。

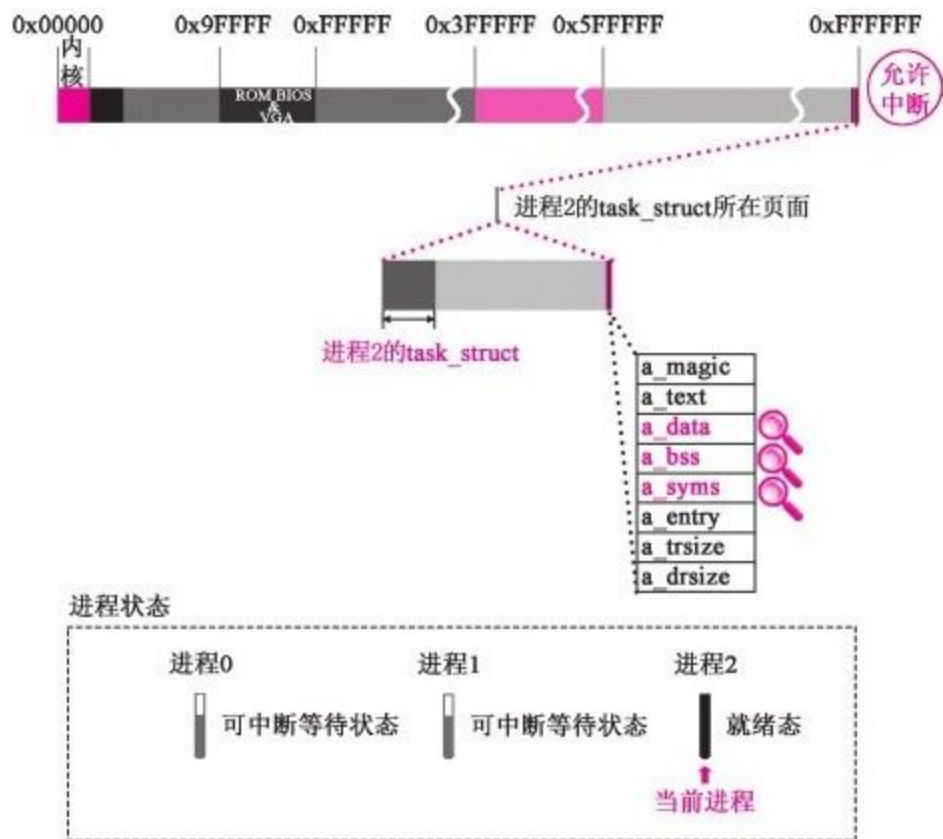


图 4-24 检测文件头属性

4.3.3 为shell程序的执行做准备

1.加载参数和环境变量

设置参数和环境变量的管理指针表page，并统计参数和环境变量个数，最终将它们复制并映射到进程2的栈空间中。

执行代码如下：

```
//代码路径： fs/exec.c:
```

```
int do_execve (unsigned long * eip,long tmp,char * filename,
```

```
char ** argv,char ** envp)
```

```
{
```

```
struct m_inode * inode;
```

```
struct buffer_head * bh;
```

```
struct exec ex;
```



```

unsigned long page[MAX_ARG_PAGES];

int i,argc,envc;

int e_uid,e_gid;

int retval;

int sh_bang=0;

unsigned long p=PAGE_SIZE * MAX_ARG_PAGES-4; //设置参数
或环境变量在进程空间的初始偏移指针

.....

for (i=0; i<MAX_ARG_PAGES; i++) /*clear page-table*/

page[i]=0; //将参数和环境变量的页面指针管理表清零

.....

argc=count (argv) ; //统计参数个数

envc=count (envp) ; //统计环境变量个数

.....

if (! sh_bang) {

    p=copy_strings (envc,envp,page,p, 0) ; //将环境变量复制到进程
空间

    p=copy_strings (argc,argv,page,p, 0) ; //将参数复制到进程空间

```

```
if (! p) {  
  
    retval=-ENOMEM;  
  
    goto exec_error2;  
  
}  
  
}  
  
.....  
  
//在进程的新栈空间中创建参数和环境变量指针管理表  
  
p= (unsigned long) create_tables ( (char *) p,argc,envc) ;  
  
.....  
  
}
```

加载参数和环境变量的情景如图4-25～图4-28所示。

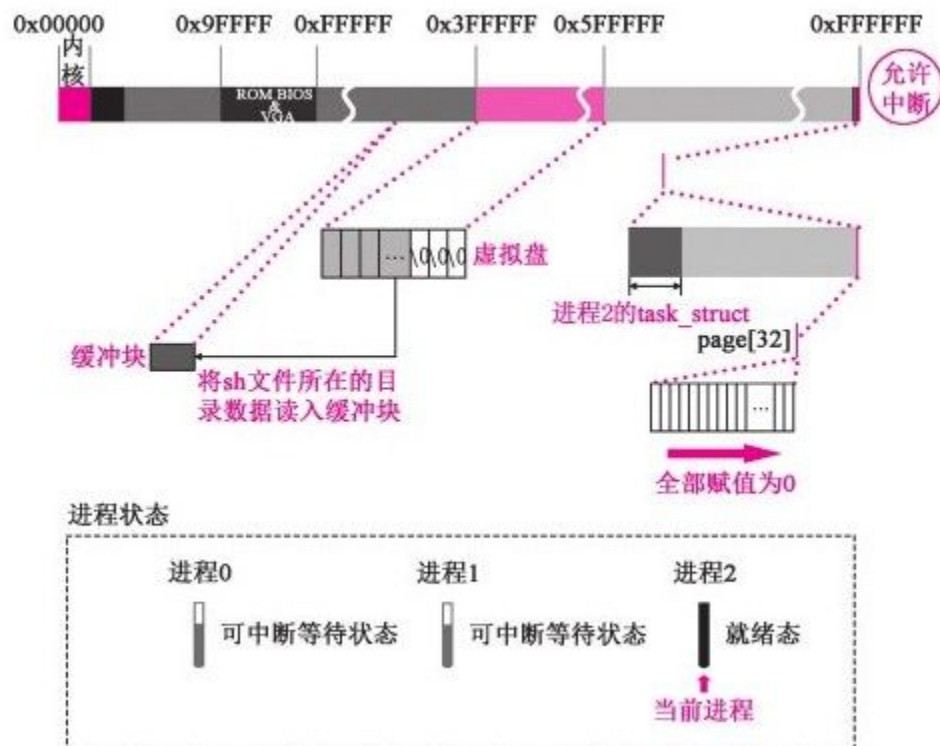


图 4-25 将参数和环境变量的载体清0

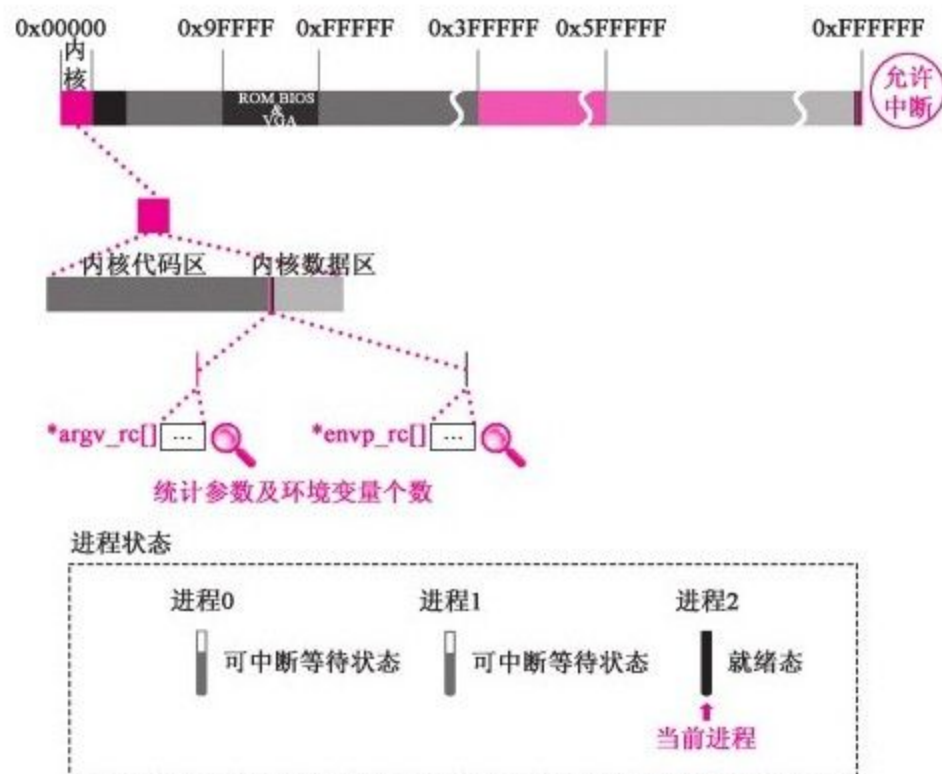


图 4-26 统计参数和环境变量个数

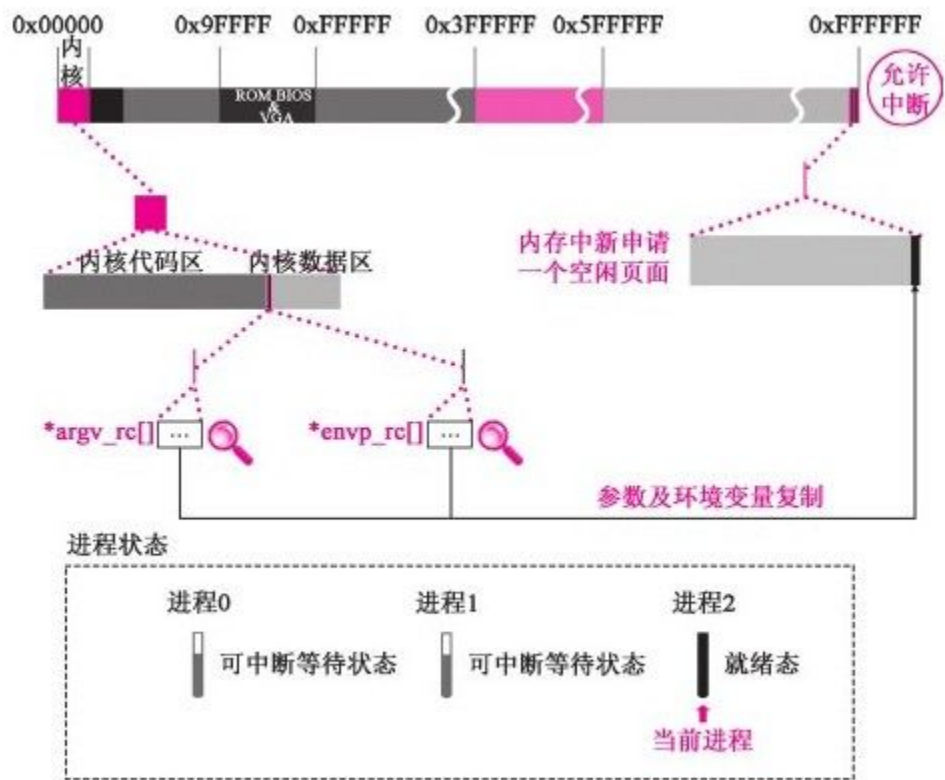


图 4-27 复制参数和环境变量

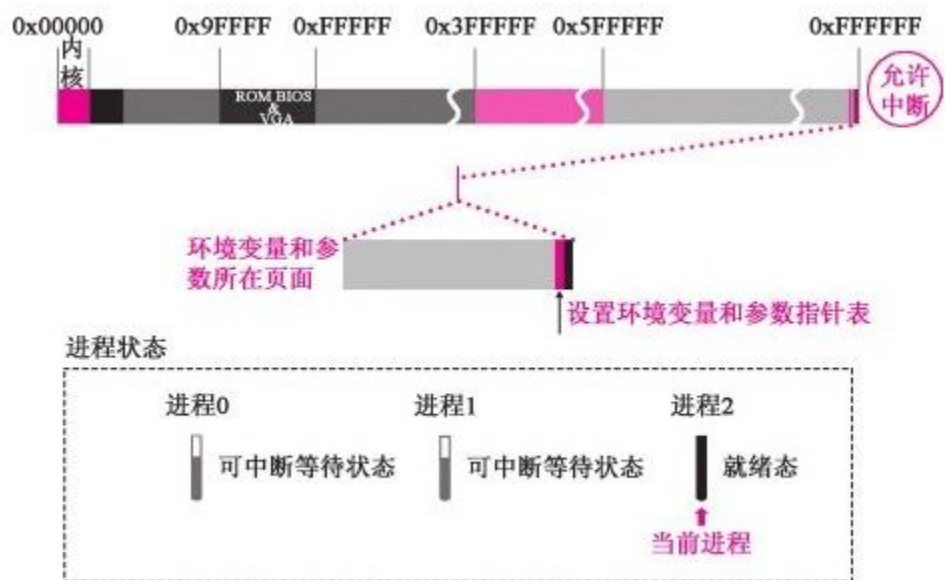


图 4-28 重新设置各段信息

2.调整进程2的管理结构

进程2有了自己对应的程序shell，因此要对自身task_struct进行调整以适应此变化。比如，原来与其父进程（进程1）共享的文件、内存页面，现在要解除关系，要根据shell程序自身情况，量身定做LDT，并设置代码段、数据段、栈段等控制变量。

代码如下：

```
//代码路径： fs/exec.c:
```

```
int do_execve (unsigned long * eip,long tmp,char * filename,
char ** argv,char ** envp)
{
```

.....

```
if (! sh_bang) {
```

```
p=copy_strings (envc,envp,page,p, 0) ;
```

```
p=copy_strings (argc,argv,page,p, 0) ;
```

```
if (! p) {
```

```
    retval=-ENOMEM;
```

```
    goto exec_error2;
```

```
}
```

```
}
```

```
/*OK,This is the point of no return*/
```

```
    if (current->executable) //检测进程是否已经有对应的可执行程序 (executable就是这个程序所在文件的i节点)
```

```
        iput (current->executable) ;
```

```
        current->executable=inode; //此时肯定没有，所以用shell程序文件的i节点设置executable
```

```
        for (i=0; i<32; i++)
```

```
            current->sigaction[i].sa_handler=NULL; //将进程2的信号管理结构全部清NULL
```

```
        for (i=0; i<NR_OPEN; i++)
```

if ((current->close_on_exec >> i) & 1) //close_on_exec所标识
的打开的文件，现在都要关闭

```
sys_close (i) ;
```

```
current->close_on_exec=0; //并将close_on_exec所有位清零
```

```
//解除进程2与进程1共享的页面关系
```

```
free_page_tables (get_base (current->ldt[1]) , get_limit  
(0x0f) ) ;
```

```
free_page_tables (get_base (current->ldt[2]) , get_limit  
(0x17) ) ;
```

```
if (last_task_used_math==current)
```

```
last_task_used_math=NULL;
```

```
current->used_math=0; //将进程2的数学协处理器的使用标志清  
零
```

```
p+=change_ldt (ex.a_text,page) -MAX_ARG_PAGES *  
PAGE_SIZE; //重新设置进程2的局部描述符表
```

```
p= (unsigned long) create_tables ( (char *) p,argc,envc) ;
```

```
current->brk=ex.a_bss+
```

```
(current->end_data=ex.a_data+
```

```
(current->end_code=ex.a_text) ) ;
```

```
current->start_stack=p&0xfffff000;
```



```
current->euid=e_uid;
```

```
current->egid=e_gid;
```

```
i=ex.a_text+ex.a_data;
```

```
while (i&0xfff)
```

```
put_fs_byte (0, (char *) (i++) ) ;
```

//设置进程代码段尾字段end_code、进程数据段尾字段end_data、进程堆结尾字段brk、栈底位置字段start_stack、有效用户ID euid和有效组ID egid，最后，再将主内存中BSS段的一页面数据全部清零

```
eip[0]=ex.a_entry; /*eip,magic happens: -) */
```

```
eip[3]=p; /*stack pointer*/
```

```
return 0;
```

```
.....
```

```
}
```

调整进程2的task_struct的情景如下图4-29～图4-33所示。

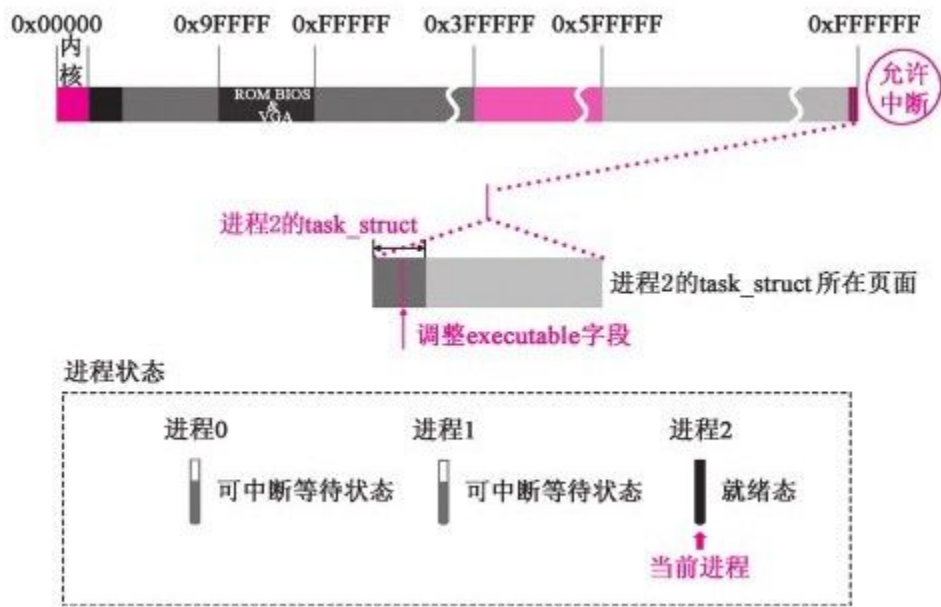


图 4-29 调整executable字段

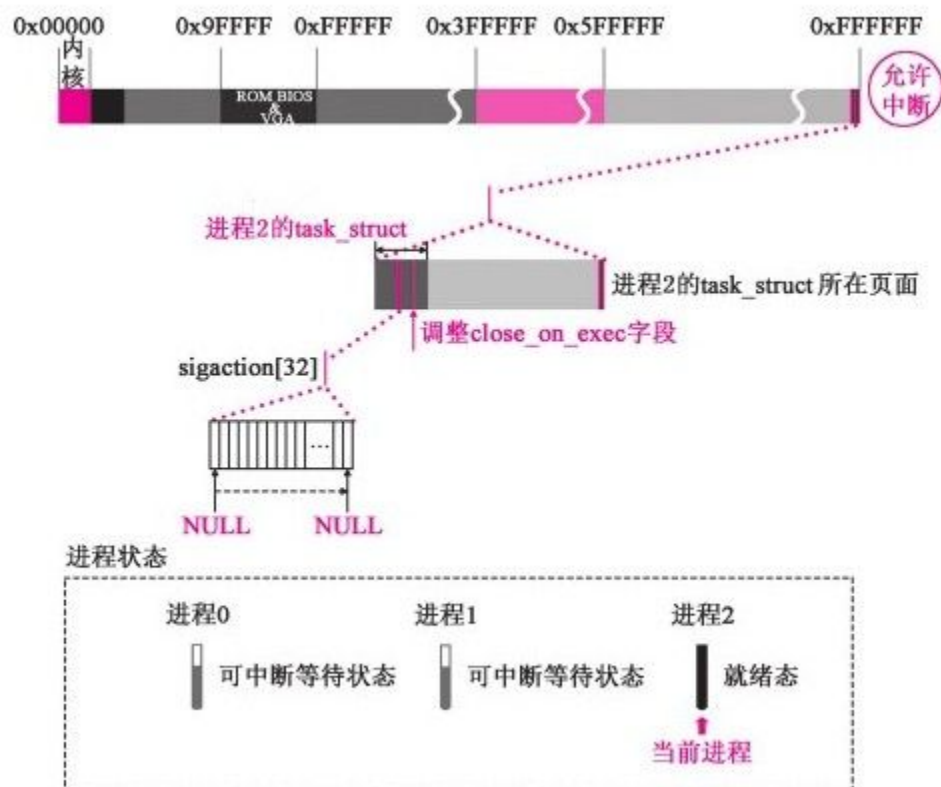


图 4-30 调整close_on_exec字段

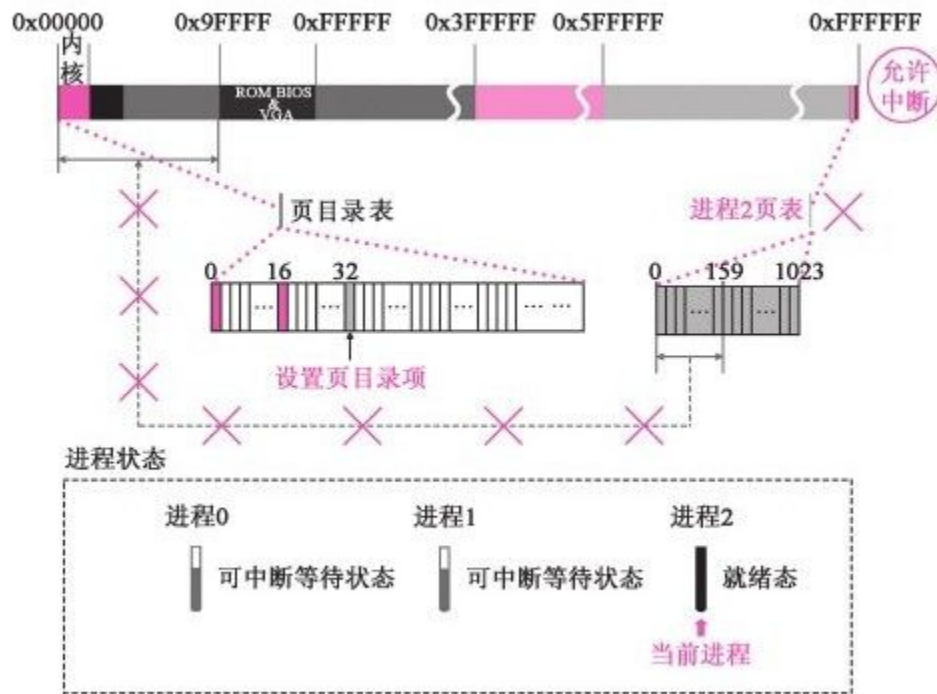


图 4-31 释放代码段与数据段所占页面



图 4-32 调整代码段与数据段段基址



图 4-32 (续)



图 4-33 调整进程2task_struct中的信息

3.为执行shell调整EIP和ESP

对sys_execve软中断压栈的值进行设置，用shell程序的起始地址值设置EIP，用进程2新的栈顶地址值设置ESP。这样，软中断iret返回后，进程2将从shell程序开始执行。代码如下：

```
//代码路径: fs/exec.c:
```

```
int do_execve (unsigned long * eip,long tmp,char * filename,
```

```
char ** argv,char ** envp)
```

```
{
```

```
.....
```

```
eip[0]=ex.a_entry; //设置进程2开始执行的EIP
```

```
eip[3]=p; //设置进程2的栈顶指针ESP
```

```
return 0;
```

```
.....
```

```
}
```

do_execve () 函数执行完毕后，sys_execve 便会中断返回，去执行shell程序。调整EIP和ESP的情景如图4-34所示。

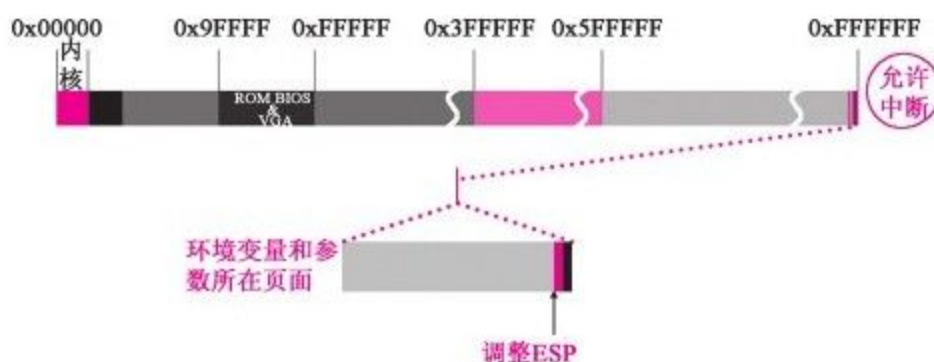


图 4-34 调整EIP和ESP

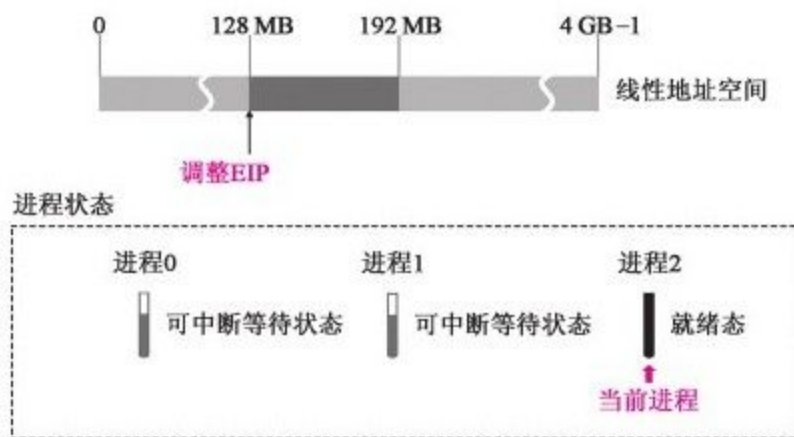


图 4-34 (续)

4.3.4 执行shell程序

1. 执行shell引导加载第一页程序

shell程序开始执行后，其线性地址空间对应的程序内容并未加载，也就不存在相应的页面，因此就会产生一个“页异常”中断。此中断会进一步调用“缺页中断”处理程序来分配该页面，并加载一页shell程序。执行代码如下：

```
//代码路径: mm/page.s:

_page_fault: //页异常处理函数入口

    xchgl %eax,    (%esp)

    pushl %ecx

    pushl %edx

    push %ds
```

```
push %es

push %fs

movl $0x10, %edx

mov %dx, %ds

mov %dx, %es

mov %dx, %fs

movl %cr2, %edx

pushl %edx

pushl %eax

testl $1, %eax

jne 1f

call _do_no_page.....//调用缺页中断处理函数

.....
```

产生缺页中断的情景如图4-35所示。

`do_no_page`（）函数开始执行后，先确定缺页的原因。假如是由于需要加载程序才缺页，会尝试与其他进程共享shell（显然此前没有进程加载过shell，无法共享），于是申请一个新的页面，并调用`bread_page`（）函数，从虚拟盘上读取4块（4 KB、一页）shell程序内容，载入内存页面。具体执行代码如下：

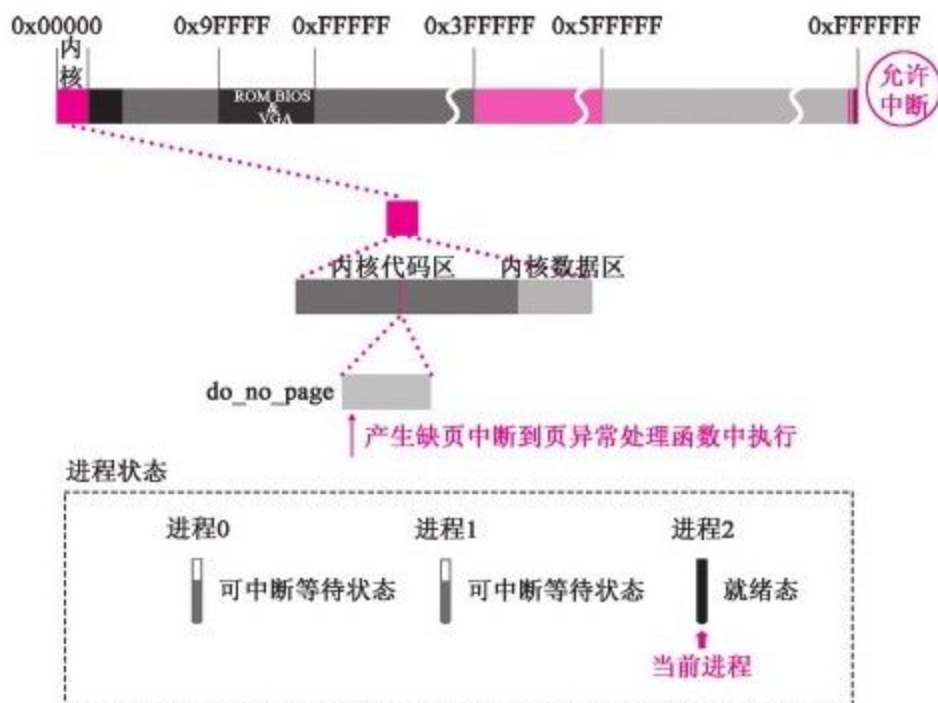


图 4-35 产生缺页中断

//代码路径: mm/memory.c:

```
void do_no_page (unsigned long error_code,unsigned long address)
```

```
{
```

```
.....int nr[4];
```

```
unsigned long tmp;
```

```
unsigned long page;
```

```
int block,i;
```

```
address&=0xfffff000;
```

```
tmp=address-current->start_code;
```

```
if (! current->executable||tmp>=current->end_data) {//如果不是  
加载程序而是其他原因导致缺页
```

```
get_empty_page (address); //比如说压栈没地方了, 那么直接申  
请页面就可以了
```

```
return; //然后直接返回
```

```
}//显然, 此时不是这种情况, 确实需要加载程序
```

```
if (share_page (tmp)) //尝试能不能和其他进程共享程序, 这样  
就省得加载了, 显然也不可能 (哪个进程也没加载过shell)
```

```
return;
```

存
if (! (page=get_free_page ())) //为shell程序申请一页新的内存

oom () ;

/*remember that 1 block is used for header*/

block=1+tmp/BLOCK_SIZE;

for (i=0; i<4; block++, i++)

nr[i]=bmap (current->executable,block) ;

bread_page (page,current->executable->i_dev,nr) ; //读取4个逻辑块 (1页) 的shell程序内容进内存页面

//在增加了一页内存后, 该页内存的部分可能会超过进程的end_data位置

//以下是对物理页面超出部分进行处理

i=tmp+4096-current->end_data;

tmp=page+4096;

while (i-->0) {

tmp--;

* (char *) tmp=0;

}

if (put_page (page,address))

```
return;

free_page (page) ;

oom () ;

}
```

申请空闲页面的情景如图4-36所示。

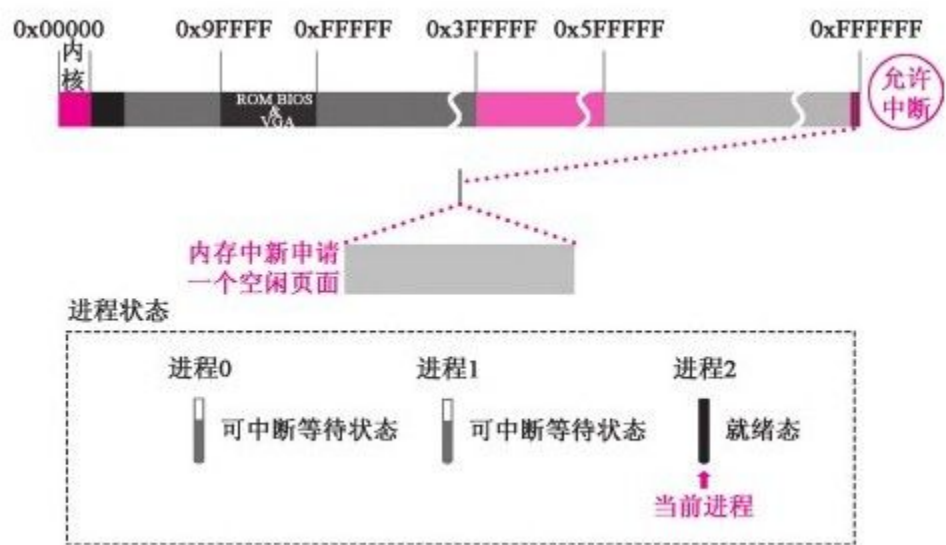


图 4-36 申请空闲页面

载入shell程序的情景如图4-37所示。

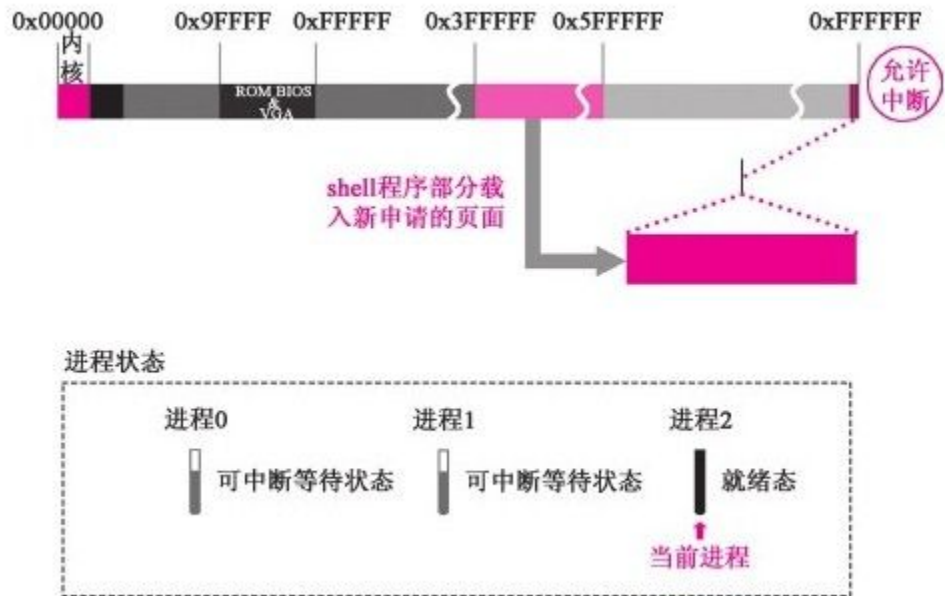


图 4-37 载入shell程序

对载入内容进行检测的情景如图4-38所示。

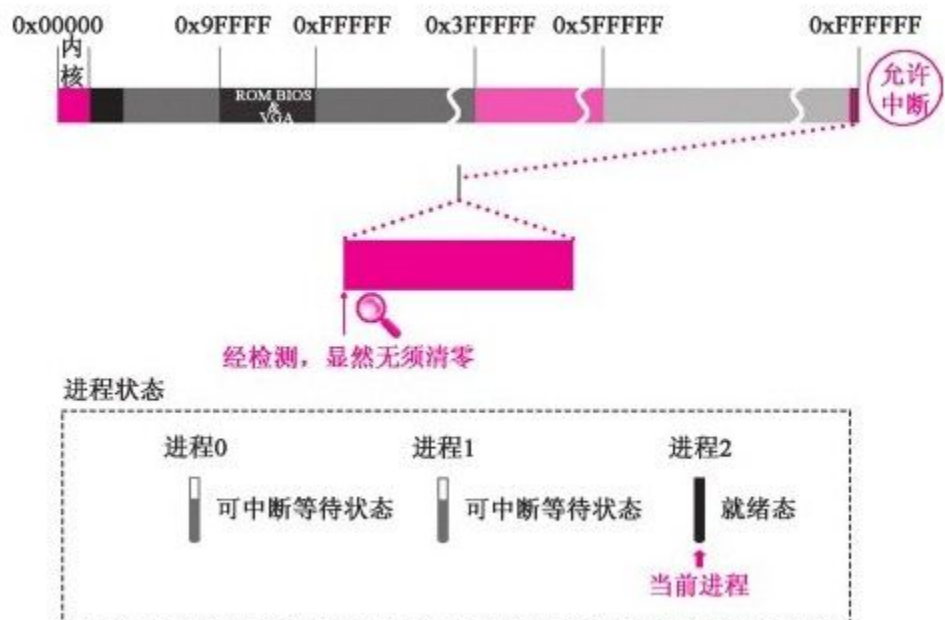


图 4-38 检测载入内容

2.映射加载页的物理地址与线性地址

载入一页的Shell程序后，内核会将该页内容映射到shell进程的线性地址空间内，建立页目录表→页表→页面的三级映射管理关系。具体执行代码如下：

//代码路径：mm/memory.c:

```
void do_no_page (unsigned long error_code,unsigned long address)
{
    .....

    put_page (page,address) //物理地址映射到线性地址空间

    .....
}
```

//代码路径：mm/memory.c:

```

unsigned long put_page (unsigned long page,unsigned long address)
{

unsigned long tmp, *page_table;

/*NOTE! This uses the fact that_pg_dir=0*/

if (page<LOW_MEM||page>=HIGH_MEMORY)

printk ("Trying to put page%p at%p\n", page,address) ;

if (mem_map[ (page-LOW_MEM) >> 12]! =1)

printk ("mem_map disagrees with%p at%p\n", page,address) ;

    page_table= (unsigned long *) ( (address>> 20) & 0xffc) ; //
计算address在页目录表中对应的表项

    if ( (*page_table) & 1) //如果该页目录项已经有对应的页表，就
获取该页表的地址

    page_table= (unsigned long *) (0xfffff000&*page_table) ;

else{//如果还没有页表，就申请一个页表

    if (! (tmp=get_free_page () ) ) //这里申请的页面用来承载页
表信息

    return 0;

    *page_table=tmp|7;

    page_table= (unsigned long *) tmp;

```

```
}
```

```
page_table[ (address >> 12) & 0x3ff]=page|7; //页面和页表建立  
关系，最终完成映射
```

```
/*no need for invalidate*/
```

```
return page;
```

```
}
```

映射的情景如图4-39所示。

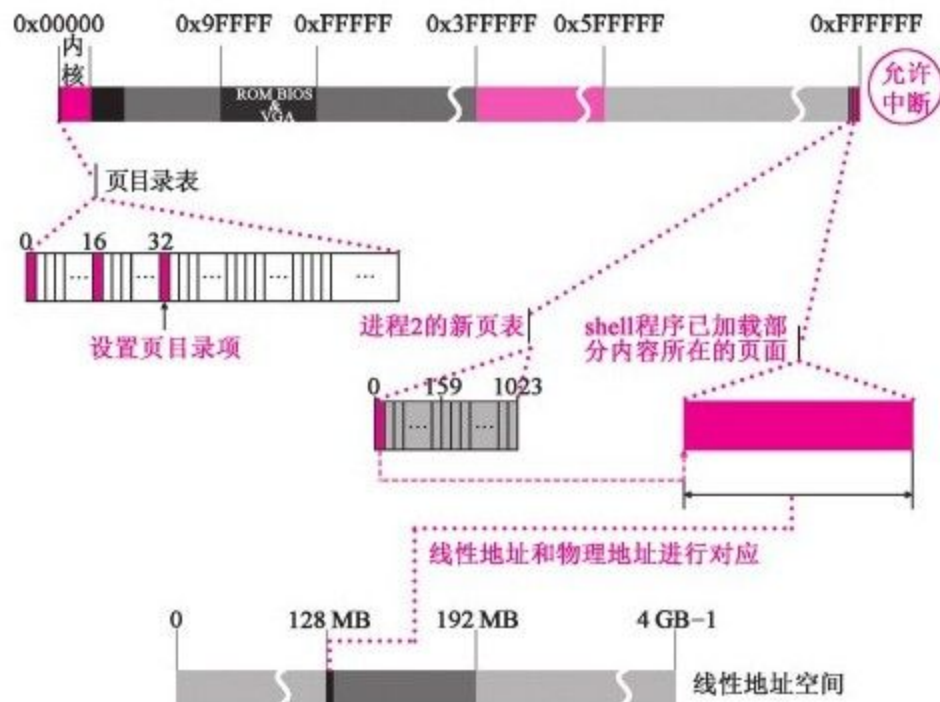


图 4-39 映射加载页的物理地址与线性地址



图 4-39 (续)

4.4 系统实现怠速

4.4.1 创建update进程

shell程序开始执行后，要读取标准输入设备文件上的信息，即task_struct中filp[20]第一项所对应文件的信息。本章4.3.1节中已经介绍到，进程2，即shell进程刚开始执行，就用rc文件替换了标准输入设备文件tty0，因此，shell程序执行后读取的是rc文件上的信息。

读取rc文件信息的情景如图4-40所示。

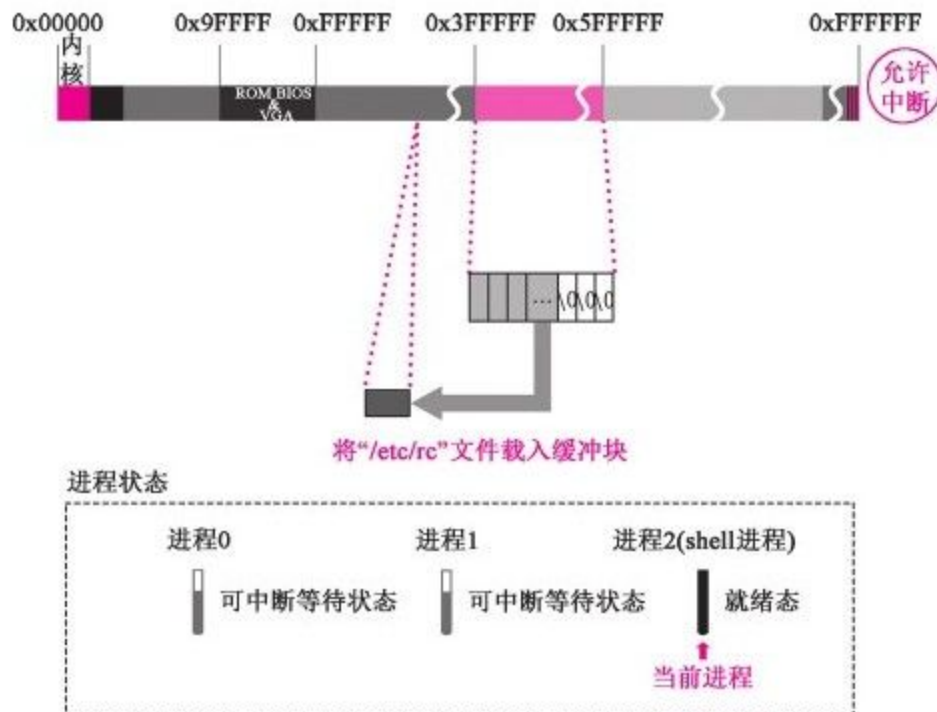


图 4-40 读取rc文件信息

shell从"/etc/rc"脚本文件中读取了一些命令，其中主要包括以下两条命令：

.....

/etc/update& //创建一个新进程，并加载update程序

.....

echo"/dev/hd1/">/etc/mtab//将"/dev/hd1/"这个字符串写入/etc/mtab文件中

.....

根据/etc/update这条命令，shell先创建一个新进程。这个新进程的进程号是3（shell进程的进程号是2，依次累加，所以它的进程号就是3）。它在task[64]中的“项号”也是3。我们在后面称之为“update进程”。创建完毕后，加载update程序，并最终将执行权转交给update进程，由它去执行。这一创建、加载、切换的过程，与本章4.2节中进程1创建进程2、切换到进程2执行以及4.3节加载shell程序的过程大体一致。

创建update进程并载入其部分程序的情景如图4-41所示。

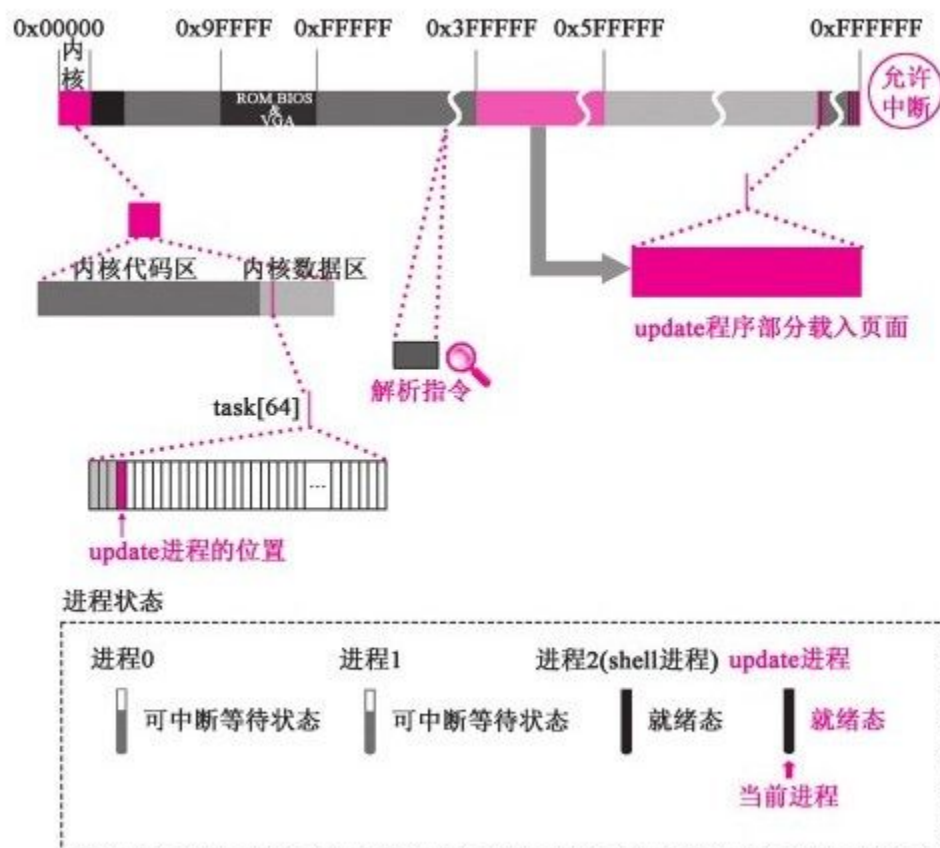


图 4-41 创建update进程并载入其部分程序

update进程有一项很重要的任务：将缓冲区中的数据同步到外设（软盘、硬盘等）上。由于主机与外设的数据交换速度远低于主机内部的数据处理速度，因此，当内核需要往外设上写数据的时候，为了提高系统的整体执行效率，并不把

数据直接写入外设上，而是先写入缓冲区，之后，根据实际情况，再将数据从缓冲区同步到外设。

每隔一段时间，`update`进程就会被唤醒，把数据往外设上同步一次，之后这个进程会被挂起，即被设置为可中断等待状态，等待着下一次被唤醒后继续执行，如此周而复始。

`update`进程执行后，并没有同步任务，于是该进程被挂起，系统进行进程调度，最终切换到`shell`进程继续执行。

切换到`shell`的情景如图4-42所示。



图 4-42 进程状态变化

4.4.2 切换到shell进程执行

4.4.1 节中介绍到，shell进程处理了rc文件中的第一条命令，创建了update进程。现在处理第二条命令，即`echo"/dev/hd1/">/etc/mtab`，将"/dev/hd1/"这一字符串写入虚拟盘中/etc/mtab文件，执行完毕后，shell程序会继续循环调用read

() 函数读取rc文件上的内容。read () 函数对应的系统调用函数是sys_read。代码如下：

//代码路径：fs/read_write.c:

```
int sys_read (unsigned int fd,char * buf,int count)
```

```
{
```

```
.....
```

```
if (inode->i_pipe) //读取管道文件
```



```

return (file->f_mode&1) ?read_pipe (inode,buf,count) : -EIO;

if (S_ISCHR (inode->i_mode) ) //读取字符设备文件

return rw_char (READ,inode->i_zone[0], buf,count, &file->
f_pos) ;

if (S_ISBLK (inode->i_mode) ) //读取块设备文件

return block_read (inode->i_zone[0], &file->f_pos,buf,count) ;

if (S_ISDIR (inode->i_mode) ||S_ISREG (inode->i_mode) )
{//读取普通文件

if (count+file->f_pos>inode->i_size)

count=inode->i_size-file->f_pos;

if (count<=0)

return 0;

return file_read (inode,file,buf,count) ;

}

printf (" (Read) inode->i_mode=%06o\n\r", inode->
i_mode) ;

return-EINVAL;

}

```

由于"/etc/rc"文件是普通文件，读取结束后，返回值应该是-ERROR（文件读取的具体操作步骤将在文件操作一章中详细介绍）。这个返回值将导致shell进程退出。退出将执行exit（）函数，对应的系统调用函数为sys_exit，执行代码如下：

```
//代码路径: kernel/exit.c:

int sys_exit (int error_code)

{

return do_exit ( (error_code&0xff) << 8) ;

}
```

进入do_exit（）函数后，开始为shell的退出做善后工作，执行代码如下：

```
//代码路径: kernel/exit.c:
```

```

int do_exit (long code)

{

int i;

    free _page_tables (get_base (current->ldt[1]) , get_limit
(0x0f) ) ; //释放shell进程代码段和数据段

    free _page_tables (get_base (current->ldt[2]) , get_limit
(0x17) ) ; //所占据的内存页面

    for (i=0; i<NR_TASKS; i++) //检测shell进程是否有子进程

    if (task[i]&&task[i]->father==current->pid) {

        //得知update进程为其子进程，因此在shell进程推出前，将update
进程的父进程

        //设置为进程1

        task[i]->father=1;

        if (task[i]->state==TASK_ZOMBIE) //如果子进程为僵死状态，
就要向进程1发送终止信号

        /*assumption task[1]is always init*/

        (void) send_sig (SIGCHLD,task[1], 1) ;

    }

    for (i=0; i<NR_OPEN; i++) //以下为解除shell进程与其他进
程、文件、终端等的关系

```

```
if (current->filp[i])

sys _close (i) ;

iput (current->pwd) ;

current->pwd=NULL;

iput (current->root) ;

current->root=NULL;

iput (current->executable) ;

current->executable=NULL;

if (current->leader&&current->tty>=0)

tty _table[current->tty].pgrp=0;

if (last_task_used_math==current)

last _task_used_math=NULL;

if (current->leader)

kill _session () ;

current->state=TASK_ZOMBIE; //将当前进程设置为僵死状态

current->exit_code=code;
```

```
tell_father (current-> father) ; //给进程1发信号，通知它shell进  
程即将推出
```

```
schedule ( ) ; //进程切换
```

```
return (-1) ; /*just to suppress warnings*/
```

```
}
```

释放页面的情景如图4-43所示。



图 4-43 释放页面

shell进程与其他进程、文件、终端.....的关系以及给父进程发送信号的情景如图4-44所示。

值得注意的是tell_father（）和schedule（）函数的执行。

tell_father（）函数执行后，会给进程1发送SIGCHLD信号，通知进程1，有子进程将要退出，执行代码如下：

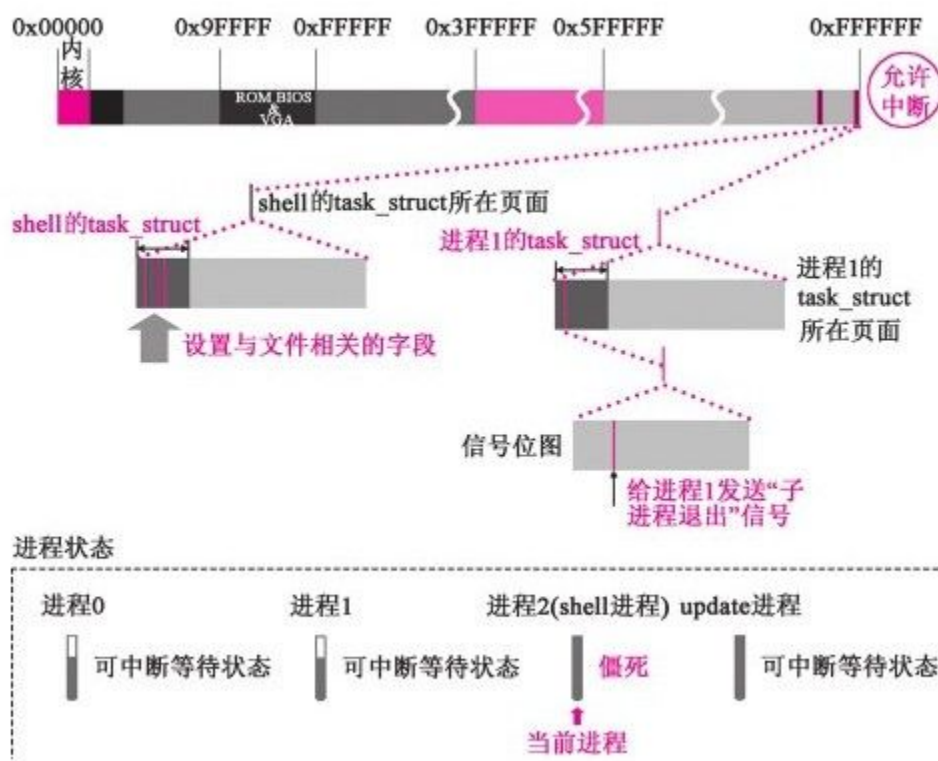


图 4-44 给父进程发送信号

//代码路径: kernel/exit.c:

static void tell_father (int pid) //通知父进程, 将有子进程退出

{

int i;

if (pid)

for (i=0; i<NR_TASKS; i++) {

if (! task[i])

continue;

if (task[i]->pid !=pid)

continue;

task[i]->signal|= (1<< (SIGCHLD-1)) ; //给父进程发送
SIGCHLD信号

return;

}

/*if we don't find any fathers,we just release ourselves*/

/*This is not really OK.Must change it to make father 1*/

printk ("BAD BAD-no father found\n\r") ;

```
release (current) ;  
  
}
```

tell_father () 函数执行完毕后，调用
schedule () 函数准备进程切换。此次schedule
() 函数中对信号的检测，影响到了进程切换。
代码如下：

```
//代码路径： kernel/sched.c:  
  
void schedule (void)  
{  
  
int i,next,c;  
  
struct task_struct ** p;  
  
/*check alarm,wake up any interruptible tasks that have got a signal*/  
  
for (p=&LAST_TASK; p> &FIRST_TASK; --p) //遍历所有进  
程  
  
if (*p) {
```



```

if ( (*p) -> alarm && (*p) -> alarm < jiffies) {

    (*p) -> signal |= (1 << (SIGALRM-1)) ;

    (*p) -> alarm=0;

}

if ( ( (*p) -> signal & ~ (_BLOCKABLE & (*p) ->
blocked) ) &&

    (*p) -> state == TASK_INTERRUPTIBLE) //发现进程1接收到了
信号并且处于可中断等待状态

    (*p) -> state = TASK_RUNNING; //将进程1设置为就绪态

}

/*this is the scheduler proper: */

while (1) {

    c=-1;

    next=0;

    i=NR_TASKS;

    p=&task[NR_TASKS];

    while (--i) {

        if (! *--p)

```

```

continue;

if ( (*p) -> state==TASK_RUNNING && (*p) -> counter > c)

c= (*p) -> counter,next=i; //发现只有进程1是就绪态

}

if (c) break;

for (p=&LAST_TASK; p> &FIRST_TASK; --p)

if (*p)

    (*p) -> counter= ( (*p) -> counter > 1) +

    (*p) -> priority;

}

switch_to (next) ; //决定切换到进程1去执行

}

```

本章4.2节中介绍到，进程1是执行sys_waitpid
 () 函数时，调用了schedule () 函数，切换到进
 程2的，所以，切换到进程1后，会继续执行

schedule () 函数，并最终回到sys_waitpid () 函数执行（具体过程参看第3章3.2节）。具体代码如下：

//代码路径: kernel/exit.c:

```
int sys_waitpid (pid_t pid,unsigned long * stat_addr,int  
options) //wait对应sys_waitpid系统调用
```

```
{
```

```
int flag,code;
```

```
struct task_struct ** p;
```

```
verify_area (stat_addr, 4) ;
```

```
repeat:
```

```
flag=0;
```

```
for (p=&LAST_TASK; p> &FIRST_TASK; --p) {
```

```
if (! *p||*p==current)
```

```
continue;
```

```
if ( (*p) -> father !=current->pid)
```

```

continue;

.....

}

if (flag) {

if (options & WNOHANG)

return 0;

current->state=TASK_INTERRUPTIBLE;

schedule () ; //执行完毕后，继续回到sys_waitpid函数中

if ( ! (current->signal &= ~ (1 << (SIGCHLD-1)) ) ) //检测到SIGCHLD信号，确定有子进程要退出

goto repeat; //重新处理子进程退出问题

else

return-EINTR;

}

return-ECHILD;

}

```

值得注意的是，此时进程1接收到的SIGCHLD信号，就是前面tell_father发送的信号。sys_waitpid（）函数的主体程序继续执行，技术路线在4.2节中已经介绍。此次执行的区别在于，确实有子进程退出，需要处理，执行代码如下：

```
//代码路径： kernel/exit.c:
```

```
int sys_waitpid (pid_t pid,unsigned long * stat_addr,int options)
```

```
{
```

```
.....
```

```
repeat:
```

```
flag=0;
```

```
for (p=&LAST_TASK; p> &FIRST_TASK; --p) {
```

```
if (! *p||*p==current)
```

```
continue;
```

```
if ( (*p) -> father !=current->pid)
```

```

continue;

.....

.....

switch ( (*p) -> state) { //继续完成shell进程退出的善后工作

.....

case TASK_ZOMBIE: //即将退出的shell为僵死状态

current->cutime+= (*p) -> utime;

current->cstime+= (*p) -> stime;

flag= (*p) -> pid; //记录进程2的进程号, "2"

code= (*p) -> exit_code;

release (*p) ; //释放shelltask_struct所在的页面

put_fs_long (code,stat_addr) ;

return flag; //返回shell进程号, "2"

.....

}

}

.....

```

}

释放shell的task_struct所在页面的情景如图4-45所示。

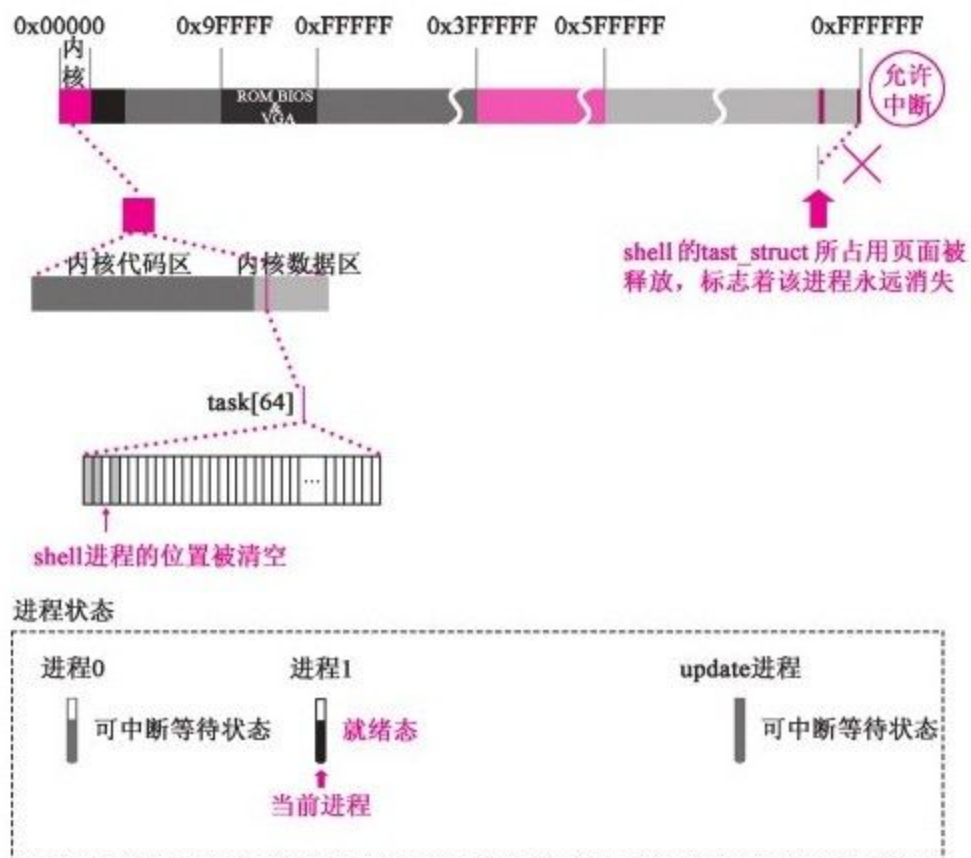


图 4-45 释放shell的task_struct所在页面

`sys_waitpid ()` 函数执行完毕后，会回到`wait ()` 函数，最终返回`init ()` 函数中，进程1继续执行，代码如下：

//代码路径： `init/main.c`:

```
void init (void)
```

```
{
```

```
.....
```

```
if (pid > 0)
```

```
while (pid != wait (&i) ) //此时执行结果为2 != 2为假，跳出循环，去下面while (1) 中执行
```

```
/*nothing*/;
```

```
while (1) { //重启shell进程
```

```
if ( (pid=fork ( ) ) < 0) {
```

```
printf ("Fork failed in init\r\n") ;
```

```
continue;
```

```
}
```



```
if (! pid) {

close (0) ; close (1) ; close (2) ;

setsid () ;

(void) open ("/dev/tty0", O_RDWR, 0) ;

(void) dup (0) ;

(void) dup (0) ;

_exit (execve ("/bin/sh", argv,envp) ) ;

}

while (1)

if (pid==wait (&i) )

break;

printf ("\n\rchild%d died with code%04x\n\r", pid,i) ;

sync () ;

}

_exit (0) ; /*NOTE! _exit,not exit () */
```

值得注意的是，本章4.2节中已经介绍到，创建完进程2后，pid值为2，而sys_waitpid返回值flag也为2，即wait（）函数返回值为2，while中条件为假，跳出循环。

4.4.3 重建shell

进程1继续执行，准备重建shell，执行代码如下：

```
//代码路径: init/main.c:

void init (void)

{

.....

if (pid > 0)

while (pid != wait (&i) ) //此时执行结果为2 != 2为假，跳出循环，去下面while (1) 中执行

/*nothing*/;

while (1) { //重启shell进程

if ( (pid=fork ()) < 0) { //进程1创建进程4，即重建shell进程

printf ("Fork failed in init\r\n") ;
```

```

    continue;

}

if (! pid) {

    close (0) ; close (1) ; close (2) ; //新的shell进程关闭所有打开的文件

    setsid () ; //创建新的会话

    (void) open ("/dev/tty0", O_RDWR, 0) ; //重新打开标准输入设备文件

    (void) dup (0) ; //重新打开标准输出设备文件

    (void) dup (0) ; //重新打开标准错误输出设备文件

    _exit (execve ("/bin/sh", argv,envp) ) ; //加载shell进程

}

while (1)

if (pid==wait (&i) ) //进程1等待子进程退出

break;

printf ("\n\rchild%d died with code%04x\n\r", pid,i) ;

sync () ;

}

```

```
_exit (0) ; /*NOTE! _exit,not exit () */
```

重建shell进程的情景如图4-46所示。

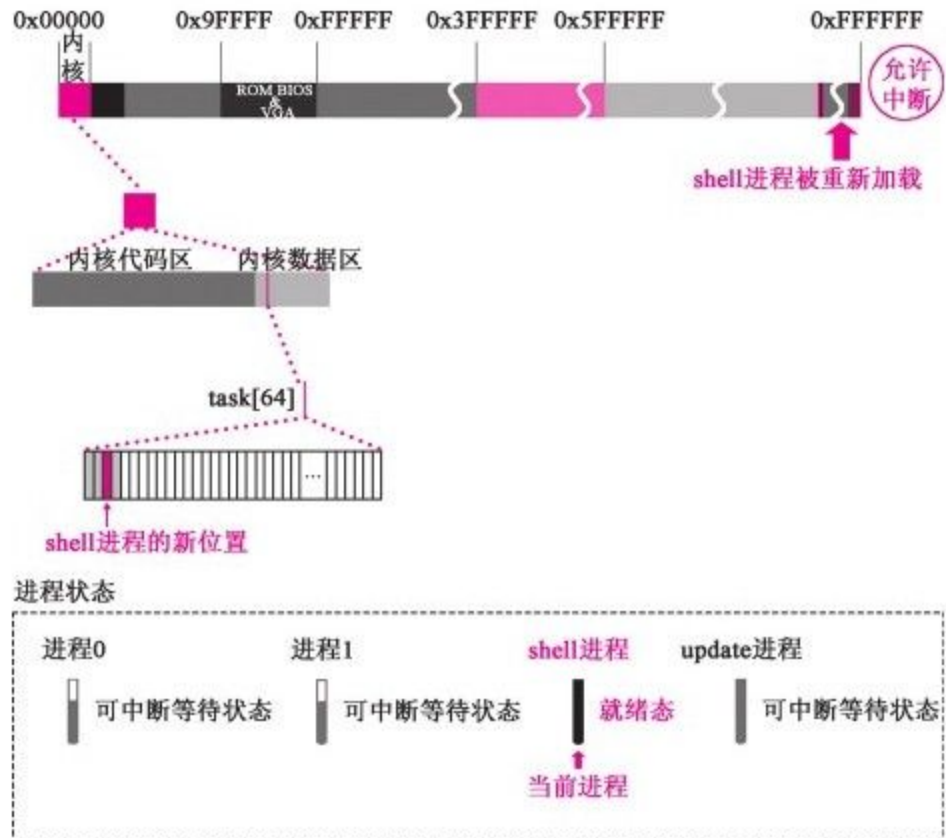


图 4-46 重建shell进程

这部分代码的执行路线，本章前面的内容中都已经介绍。值得注意的区别是，`shell`进程的进

程号由last_pid累加产生，因此为4，但它占用的task[64]的项，是前面已退出的shell进程的项，因此项号仍然是2。另外，此次shell重新打开标准输入设备文件（tty0文件），而非rc文件，这使得shell开始执行后，不再退出。执行代码如下：

```
//代码路径： fs/read_write.c:
```

```
int sys_read (unsigned int fd,char * buf,int count)

{

.....

if (inode->i_pipe)

return (file->f_mode&1) ?read_pipe (inode,buf,count) : -EIO;

if (S_ISCHR (inode->i_mode) ) //此次shell读取的tty0文件为字符设备文件

return rw_char (READ,inode->i_zone[0], buf,count, &file->f_pos) ;

if (S_ISBLK (inode->i_mode) )
```

```

    return block_read (inode->i_zone[0], &file->f_pos,buf,count) ;

    if (S_ISDIR (inode->i_mode) ||S_ISREG (inode->i_mode) )
    { //上次shell读取的rc文件为普通文件

        if (count+file->f_pos > inode->i_size)

            count=inode->i_size-file->f_pos;

        if (count <=0)

            return 0;

        return file_read (inode,file,buf,count) ;

    }

    printk (" (Read) inode->i_mode=%06o\n\r", inode->
i_mode) ;

    return-EINVAL;

}

```

在进入rw_char () 函数后，shell进程将被设置为可中断等待状态，这样所有的进程全部都处

于可中断等待状态，再次切换到进程0去执行，系统实现怠速。

怠速以后，操作系统用户将通过shell进程提供的平台与计算机进行交互。shell进程处理用户指令的工作原理如下：用户通过键盘输入信息，存储在指定的字符缓冲队列上。该缓冲队列上的内容，就是tty0文件的内容。shell进程会不断读取缓冲队列上的数据信息。如果用户没有下达指令，缓冲队列中就不会有数据，shell进程将会被设置为可中断等待状态，即被挂起。如果用户通过键盘下达指令，将产生键盘中断，中断服务程序会将字符信息存储在缓冲队列上，并给shell进程发信号，信号将导致shell进程被设置为就绪状态，即被唤醒，唤醒后的shell继续从缓冲队列中

读取数据信息并处理，完毕后，`shell`进程将再次被挂起，等待下一次键盘中断被唤醒。

4.5 本章小结

本章详细讲解了进程1创建进程2、加载shell程序、创建update进程、重建shell、实现系统怠速的全过程。与前面讲解激活进程0、进程0重建进程1有较大不同的地方是，进程0和进程1的代码都是操作系统设计者直接写到内核代码中的，进程2则是从硬盘中加载的执行代码，而且shell进程是对操作系统非常重要的用户界面进程。所以，本章内容涉及的打开终端设备文件以及大量的文件操作，为后续学习文件操作打下了基础。

第5章 文件操作

本章将通过几个实例程序详细讲解操作系统的文件操作。

5.1 安装文件系统

在3.3.3节中，操作系统成功加载了根文件系统，使之能够以文件的形式与根设备进行数据交互。安装文件系统就是在根文件系统的基础上，把硬盘中的文件系统安装在根文件系统上，使操作系统也具备以文件的形式与硬盘进行数据交互的能力。

安装文件系统分为三步：

1) 将硬盘上的超级块读取出来，并载入系统中的super_block[8]中。

2) 将虚拟盘上指定的i节点读出，并将此i节点加载到系统中的inode_table[32]中。

3) 将硬盘上的超级块挂接到inode_table[32]中指定的i节点上。

硬盘的文件系统安装成功后，整体结构关系如图5-1所示。

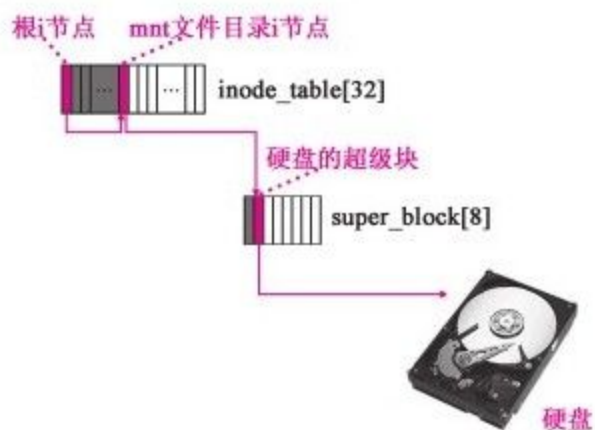


图 5-1 硬盘文件系统安装成功后的示意图

在shell下输入“mount/dev/hd1/mnt”命令来安装文件系统。此命令包括三个参数，分别是“mount”、“/dev/hd1”和“/mnt”。“mount”是这个命令的名字，表明这个命令是要安装文件系统；“/dev/hd1”和“/mnt”是两个路径名。整条命令的意思是：将设备“hd1”的文件系统挂载在“mnt”目录文件下。shell进程接到该命令后，会创建一个新进程，新进程调用mount（）函数，并最终映射到sys_mount（）系统调用函数。安装文件系统的工作就是由sys_mount（）函数完成的。

5.1.1 获取外设的超级块

小贴士

硬盘是可以分区的，每个分区都可以算作一个设备。本章和以后的章节中，默认整个硬盘就是一个分区，因此hd1就代表了硬盘这个设备。

`sys_mount`（）函数先调用`namei`（）函数，根据`/dev/hd1`路径名，获得hd1设备文件的i节点，再从i节点中获得设备号，最终根据设备号，读取设备的超级块。

执行代码如下：

```
//代码路径： fs/super.c:
```

```
int sys_mount (char * dev_name,char * dir_name,int rw_flag)
```

```
{
```

```
    struct m_inode * dev_i, *dir_i;
```

```
    struct super_block * sb;
```

```
    int dev;
```

```

if ( ! (dev_i=namei (dev_name) ) ) //获取hd1设备文件i节点

return-ENOENT;

dev=dev_i->i_zone[0]; //通过i节点，获取设备号

if ( ! S_ISBLK (dev_i->i_mode) ) {

//如果hd1文件不是块设备文件

iput (dev_i) ; //就释放掉它的i节点

return-EPERM;

}

iput (dev_i) ; //释放hd1设备文件i节点

if ( ! (dir_i=namei (dir_name) ) )

return-ENOENT;

if (dir_i->i_count ! =1||dir_i->i_num==ROOT_INO) {

iput (dir_i) ;

return-EBUSY;

}

if ( ! S_ISDIR (dir_i->i_mode) ) {

iput (dir_i) ;

```

```
return-EPERM;
```

```
}
```

```
if (! (sb=read_super (dev) ) ) { //通过设备号，读取设备的超  
级块
```

```
iput (dir_i) ;
```

```
return-EBUSY;
```

```
}
```

```
if (sb->s_imount) {
```

```
iput (dir_i) ;
```

```
return-EBUSY;
```

```
}
```

```
if (dir_i->i_mount) {
```

```
iput (dir_i) ;
```

```
return-EPERM;
```

```
}
```

```
sb->s_imount=dir_i;
```

```
dir_i->i_mount=1;
```



```
dir_i->i_dirt=1; /*NOTE! we don't iput (dir_i) */  
  
return 0; /*we do that in umount*/  
  
}
```

其中namei () 函数获取i节点的过程与4.1.1中介绍的i节点获取过程大体一致。read_super () 函数读取设备超级块大体分为三步：第一，在super_block中选定一个空闲项来存储超级块；第二，将超级块载入该项；第三，根据超级块中提供的信息，载入i节点位图和逻辑块位图。另外，在操作超级块表项前要将其加锁，以免被其他操作干扰，等操作完毕后再解锁。此过程的执行代码如下：

//代码路径：fs/super.c:

```
static struct super_block * read_super (int dev)
```

```

{

struct super_block * s;

struct buffer_head * bh;

int i,block;

if (! dev)

return NULL;

check_disk_change (dev) ;

if (s=get_super (dev) ) //如果hd1设备的超级块已经载入，则直接返回

return s;

for (s=0+super_block; s++) { //在super_block中为hd1超级块寻找空闲位置

if (s >= NR_SUPER+super_block)

return NULL;

if (! s->s_dev) //确定super_block第二项为空闲位置

break;

}

s->s_dev=dev; //以下的s->..是对该超级块项中与内存操作相关部分进行设置

```

```
s->s_isup=NULL;
```

```
s->s_imount=NULL;
```

```
s->s_time=0;
```

```
s->s_rd_only=0;
```

```
s->s_dirt=0;
```

```
lock_super (s) ; //对该超级块项加锁保护，以免设置过程中被  
干扰
```

```
//根据hd1设备号和块号（1，代表设备上第二块，即超级块所在逻辑  
块号），读取超级块
```

```
if ( ! (bh=bread (dev, 1) ) ) {
```

```
s->s_dev=0;
```

```
free_super (s) ;
```

```
return NULL;
```

```
}
```

```
* ( (struct d_super_block *) s) =//将读取的超级块信息载入超级  
块项中，即第二项中
```

```
* ( (struct d_super_block *) bh->b_data) ;
```

```
brelse (bh) ;
```

```
    if (s->s_magic != SUPER_MAGIC) { //检测超级块魔数，确定设  
备的文件系统是否可用
```

```
    s->s_dev=0;
```

```
    free_super (s) ;
```

```
    return NULL;
```

```
}
```

```
    //以下是载入i节点位图和超级块位图，并与超级块中s_imap和  
s_zmap建立对应关系
```

```
    for (i=0; i<I_MAP_SLOTS; i++)
```

```
    s->s_imap[i]=NULL;
```

```
    for (i=0; i<Z_MAP_SLOTS; i++)
```

```
    s->s_zmap[i]=NULL;
```

```
    block=2;
```

```
    for (i=0; i<s->s_imap_blocks; i++)
```

```
    if (s->s_imap[i]=bread (dev,block) )
```

```
    block++;
```

```
    else
```

```
    break;
```

```
for (i=0; i<s->s_zmap_blocks; i++)
```

```
if (s->s_zmap[i]=bread (dev,block) )
```

```
block++;
```

```
else
```

```
break;
```

//检查从设备上读取的逻辑块数block是否与其应有的数量2+s->s_imap_blocks+s->s_zmap_blocks相一致

```
if (block!=2+s->s_imap_blocks+s->s_zmap_blocks) {
```

```
for (i=0; i<I_MAP_SLOTS; i++)
```

```
breakelse (s->s_imap[i]) ;
```

```
for (i=0; i<Z_MAP_SLOTS; i++)
```

```
breakelse (s->s_zmap[i]) ;
```

```
s->s_dev=0;
```

```
free _super (s) ;
```

```
return NULL;
```

```
}
```

```
s->s_imap[0]->b_data[0]=1;
```

```
s->s_zmap[0]->b_data[0]=1;
```

```
free _super (s) ; //超级块设置完毕，解除对超级块项的保护
```

```
return s;
```

```
}
```

5.1.2 确定根文件系统的挂接点

再次调用namei（）函数，根据/mnt路径名，获得mnt目录文件的i节点，获取后，分析i节点的属性，判断该i节点是否可以用来挂接文件系统，执行代码如下：

//代码路径： fs/super.c:

```
int sys_mount (char * dev_name,char * dir_name,int rw_flag)
{
    struct m_inode * dev_i, *dir_i;

    struct super_block * sb;

    int dev;

    if ( ! (dev_i=namei (dev_name) ) ) //获取hd1设备文件i节点
        return-ENOENT;

    dev=dev_i->i_zone[0]; //通过i节点，获取设备号
```

if (! S_ISBLK (dev_i->i_mode)) { //如果hd1文件不是块设备文件

iput (dev_i) ; //就释放掉它的i节点

return-EPERM;

}

iput (dev_i) ; //释放hd1设备文件i节点

if (! (dir_i=namei (dir_name))) //获取mnt目录文件i节点

return-ENOENT;

//确定mnt的i节点只被引用过一次，而且不是根i节点，它才能被使用

if (dir_i->i_count ! =1||dir_i->i_num==ROOT_INO) {

iput (dir_i) ;

return-EBUSY;

}

if (! S_ISDIR (dir_i->i_mode)) { //确定mnt确实是目录文件

iput (dir_i) ;

return-EPERM;

}

if (! (sb=read_super (dev))) { //通过设备号，获取设备的超级块

iput (dir_i) ;

return-EBUSY;

}

if (sb->s_imount) {

iput (dir_i) ;

return-EBUSY;

}

if (dir_i->i_mount) {

iput (dir_i) ;

return-EPERM;

}

sb->s_imount=dir_i;

dir_i->i_mount=1;

dir_i->i_dirt=1; /*NOTE ! we don't iput (dir_i) */

return 0; /*we do that in umount*/

}

经分析确定，`mnt`具备挂接文件系统的条件。

5.1.3 将超级块与根文件系统挂接

挂接前要确保挂接点和被挂接点都是“干净”的，即hd1设备的文件系统没有被安装过，而且mnt目录文件上也没有安装过其他文件系统。这两点确定后，就将两者挂接，执行代码如下：

//代码路径：fs/super.c:

```
int sys_mount (char * dev_name,char * dir_name,int rw_flag)
{
    .....

    if ( ! (sb=read_super (dev) ) ) { //通过设备号，获取设备的超
级块

        iput (dir_i) ;

        return-EBUSY;

    }
```

if (sb->s_imount) { //确保hd1设备的文件系统没有被安装在其他地方

input (dir_i) ;

return-EBUSY;

}

if (dir_i->i_mount) { //确保mnt目录文件i节点上没有安装过其他文件系统

input (dir_i) ;

return-EPERM;

}

sb->s_imount=dir_i; //将超级块中s_imount与根文件系统中dir_i挂接

dir_i->i_mount=1; //给dir_i做标记，表明该i节点上已经挂接了文件系统

dir_i->i_dirt=1; //给dir_i做标记，表明i节点上的信息已经被更改

/*NOTE! we don't input (dir_i) */

return 0; /*we do that in umount*/

}

本章第2至8节，将通过3个文件操作实例，讲解文件系统的工作原理。

实例1：用户进程打开一个硬盘上已存在的文件，并读取文件的内容。

实例2：用户进程在硬盘上新建一个文件，并将内容写入这个文件。

实例3：关闭此文件，并将其从硬盘中删除。

实例1：用户进程打开一个在硬盘上已存在的文件，并读取文件的内容。

本实例分为两部分：打开文件和读取文件，进程的代码如下：

```
void main ()
```

```
{  
  
//打开文件  
  
char buffer[12000];  
  
int fd=open ("/mnt/user/user1/user2/hello.txt", O_RDWR,  
0644) ) ;  
  
//读取文件  
  
int size=read (fd,buffer,sizeof (buffer) ) ;  
  
return;  
  
}
```

5.2 打开文件

打开文件是拟人化的表述，在操作系统中就是确定进程操作哪个文件。这个确定过程由两件事构成：

- 1) 将用户进程task_struct中的*filp[20]与内核中的file_table[64]进行挂接。
- 2) 将用户进程需要打开的文件对应的i节点在file_table[64]中进行登记。

操作系统根据用户进程的需求来操作文件，内核通过*filp[20]掌控一个进程可以打开的文件，既可以打开多个不同的文件，也可以同一个文件多次打开，每打开一次文件（不论是否是同一个

文件），就要在*filp[20]中占用一个项（比如hello.txt文件被一个用户进程打开两次，就要在*filp[20]中占用两项）记录指针，所以，一个进程可以同时打开的文件次数不能超过20次。

操作系统中file_table[64]是管理所有进程打开文件的数据结构，不但记录了不同的进程打开不同的文件，也记录了不同的进程打开同一个文件，甚至记录了同一个进程多次打开同一个文件。与*filp[20]类似，只要打开一次文件，就要在file_table[64]中记录。

文件的i节点是记载文件属性的最关键的数据结构。在操作系统中i节点和文件是一一对应的，找到i节点就能唯一确定文件。内核通过

inode_table[32]掌控正在使用的文件i节点数，每个被使用的文件i节点都要记录在其中。

打开文件的本质就是要建立*filp[20]、file_table[64]、inode_table[32]三者之间的关系，如图5-2所示。

这个过程分为三个步骤进行：

第一步，将用户进程task_struct中的*filp[20]与内核中的file_table[64]进行挂接。

第二步，以用户给定的路径名“/mnt/user/user1/user2/hello.txt”为线索，找到hello.txt文件的i节点。

第三步，将hello.txt对应的i节点在file_table[64]中进行登记。

具体的操作是在进程中调用open（）函数实现打开文件，该函数最终映射到sys_open（）系统调用函数执行。映射过程以及sys_open（）函数的基本执行情况已经在本书4.4.1节中介绍。本节在此基础上，详细讲解sys_open（）函数的执行细节并分析sys_open（）的设计思路。

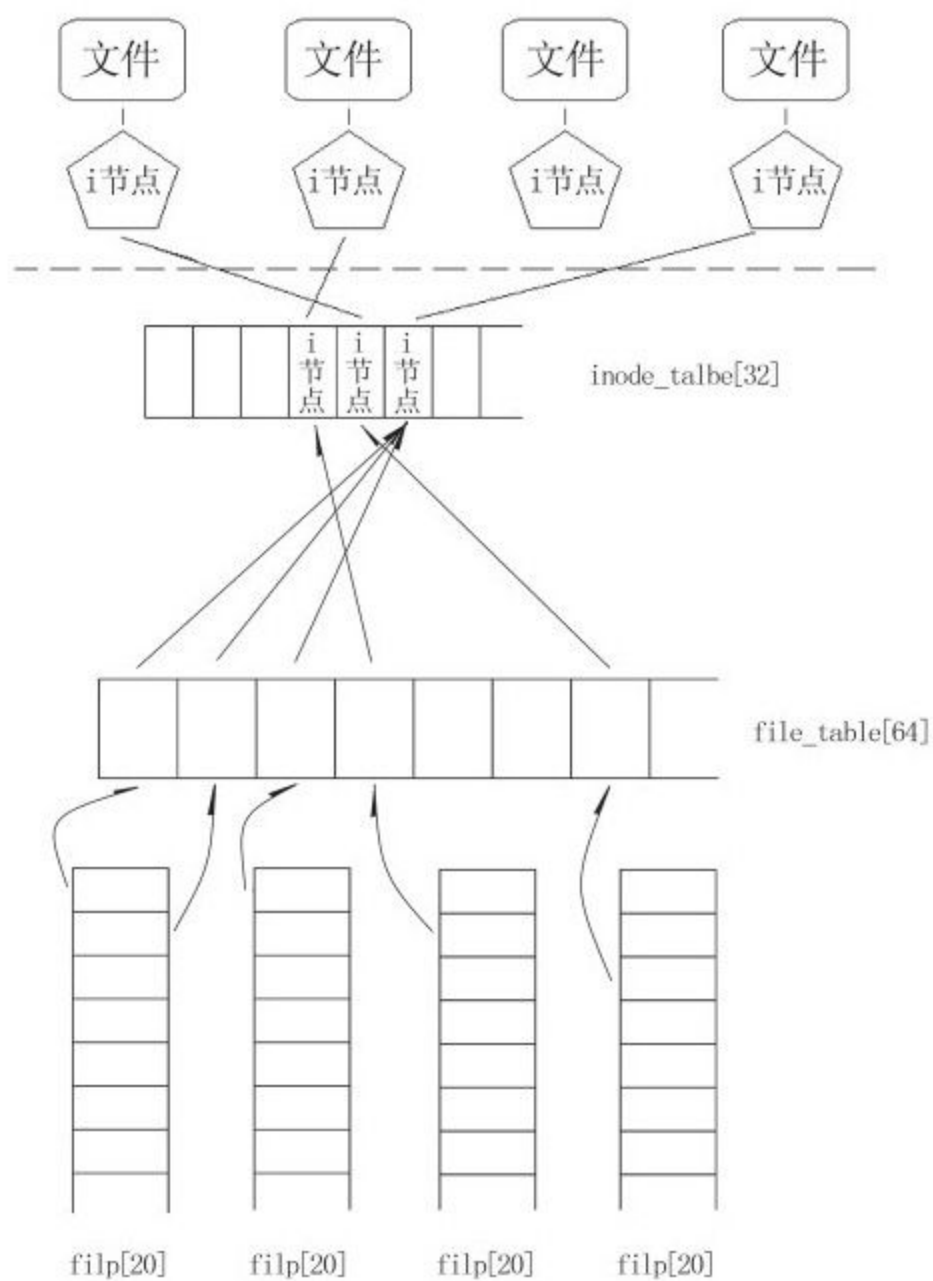


图 5-2 打开文件的关系示意图

5.2.1 将进程的*filp[20]与file_table[64]挂接

在sys_open（）函数中，实现*filp[20]与file_table[64]进行挂接任务的代码如下： </p>

```
//代码路径： include/linux/fs.h:

#define NR_OPEN 20//进程可以打开文件的最大数

#define NR_FILE 64//操作系统可以打开文件的最大数

.....

struct file{

    unsigned short f_mode; //文件操作模式

    unsigned short f_flags; //文件打开、控制标志

    unsigned short f_count; //文件句柄数

    struct m_inode * f_inode; //指向文件对应的i节点

    off_t f_pos; //文件位置（读写偏移值）
```

```
};
```

```
//代码路径: include/linux/sched.h:
```

```
struct file * filp[NR_OPEN]; //管理进程使用文件的指针数组
```

```
//代码路径: fs/file_table.c:
```

```
struct file file_table[NR_FILE]; //记录操作系统已经打开文件的管  
控信息
```

```
//代码路径: fs/open.c:
```

```
int sys_open (const char * filename,int flag,int mode)
```

```
{
```

```
struct m_inode * inode;
```

```
struct file * f;
```

```
int i,fd;
```

```
mode&=0777&~current->umask; //设置该文件模式为用户许可  
使用模式 (将在5.4新建文件一节中讲解)
```

```
for (fd=0; fd<NR_OPEN; fd++) //从当前进程*filp[20]中寻找空  
闲项
```

```
if (! current->filp[fd])
```

```
break;
```

```
if (fd>=NR_OPEN) //检查*filp[20]结构是否已经超出使用极限
```

```

return-EINVAL;

current->close_on_exec&=~ (1<<fd) ; //将该文件句柄执行时
关闭标志设置为0（将在第6章中讲解）

f=0+file_table;

for (i=0; i<NR_FILE; i++, f++) //在file_table[64]中寻找空闲
项

if (! f->f_count) break;

if (i>=NR_FILE) //检查file_table[64]结构是否已经超出使用极限
（极限为承载64个文件表项）

return-EINVAL;

(current->filp[fd]=f) ->f_count++; //将当前进程的*filp[20]与
file_table[64]对应项挂接，并增加文件句柄计数

.....

}

```

要想实现挂接任务，就要分别在*filp[20]、file_table[64]中找到空闲项，找到后，将当前进程的*filp[20]与file_table[64]的对应项挂接，并增加

`file_table[64]`中对应项的文件句柄计数（文件句柄将在第7章讲解）。

另外需要注意的是，多个进程对文件的使用情况是错综复杂的，无法事先预计，在找寻 `*filp[20]`、`file_table[64]`的空闲项时不一定都能找得到，也就是说，超出了两个数据结构的使用极限，遇到这类情况内核会给出错误信息。在文件系统中先检查、后使用的设计风格是贯穿始终的。

5.2.2 获取文件i节点

本节通过分析路径名“/mnt/user/user1/user2/hello.txt”，找到hello.txt文件的i节点。

此次与本书4.1.1节中所介绍的查找文件i节点的区别在于，hello.txt文件存储于硬盘上，查找过程将会从根i节点开始，通过虚拟盘找到硬盘上的文件。查找的技术路线如图5-3所示。

从图5-3中可以看出，路径名的解析过程有明显的同构性，技术路线是：

寻找i节点 → 通过i节点找到目录文件 →

通过目录文件找到目录项 → 通过目录项找到
目录文件i节点号 →

通过目录文件找到目录项 → 通过目录项找到
目录文件i节点号 →

.....

循环往复，最终找到hello.txt文件。

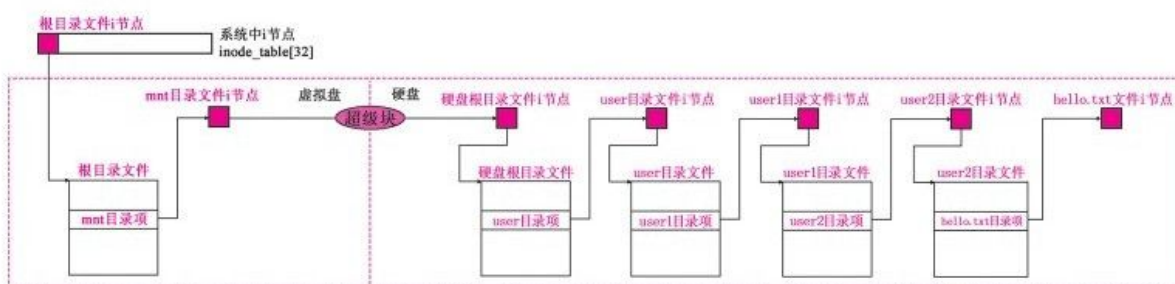


图 5-3 文件路径解析示意图

1. 获取目录文件i节点

获取i节点的程序调用图如图5-4所示。

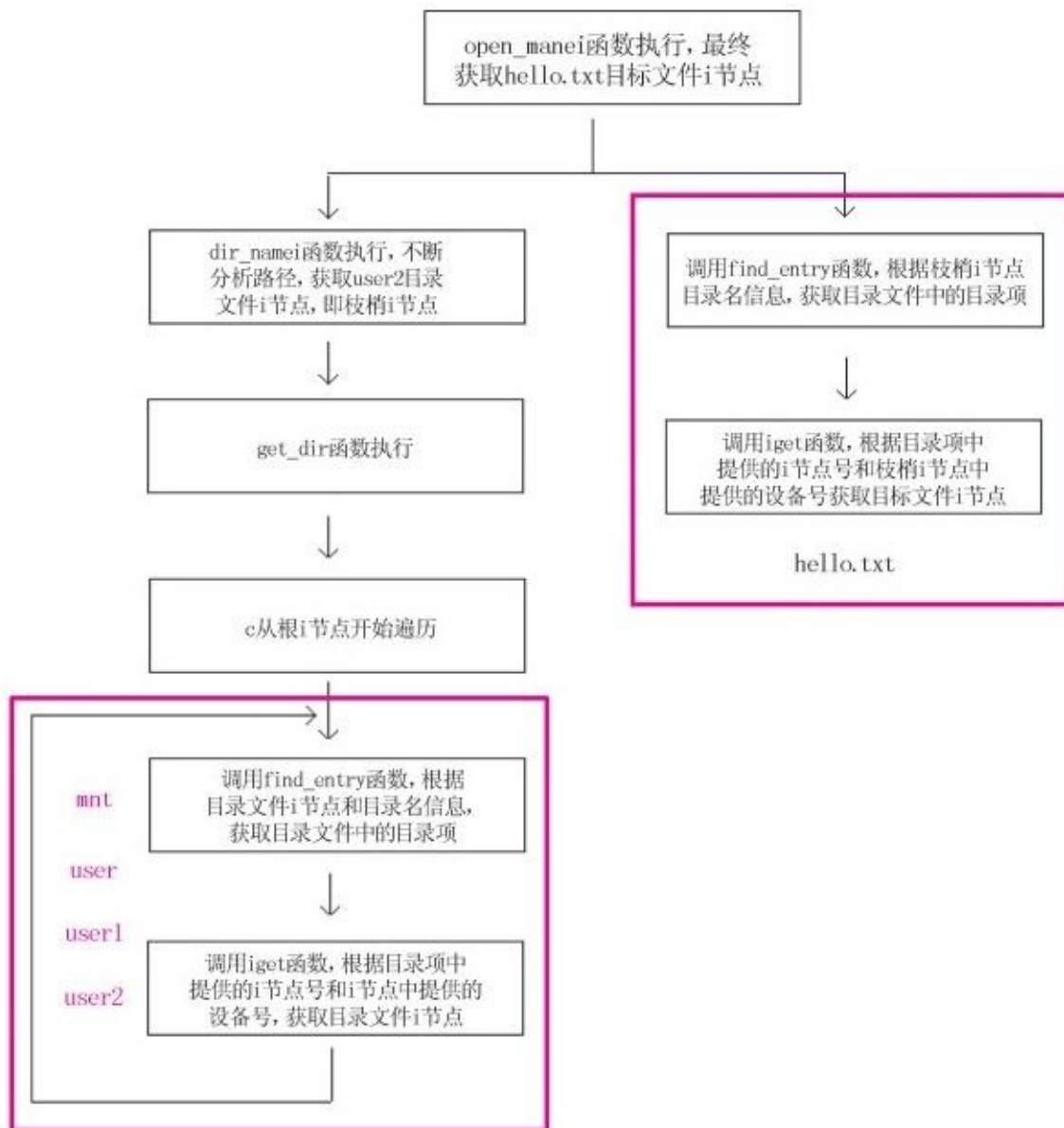


图 5-4 获取i节点的程序调用图

获取目录文件i节点是通过调用open_namei

() 函数实现的。代码如下：

//代码路径: fs/open.c:

int sys_open (const char * filename,int flag,int mode)

{

.....

if ((i=open_namei (filename,flag,mode, &inode)) < 0) { //获取
hello.txt文件i节点

current->filp[fd]=NULL; //如果没有获取到i节点, 将*filp[20]申请
到的表项置为NULL

f->f_count=0; //如果没有获取到i节点, 将file_table[64]申请到的
表项引用计数置0

return i;

}

.....

}

进入open_namei () 函数后，先对所要打开的文件按照用户需求设置参数flag、mode。

执行代码如下：

```
//代码路径：include/fcntl.h: //八进制形式：

.....

#define O_ACCMODE 00003//文件访问模式屏蔽码

#define O_RDONLY 00//只读方式打开文件标志

#define O_WRONLY 01//只写方式打开文件标志

#define O_RDWR 02//读写方式打开文件标志

#define O_CREAT 00100/*not fcntl*///创建新文件标志

#define O_EXCL 00200/*not fcntl*///进程独占文件标志

#define O_NOCTTY 00400/*not fcntl*///不分配控制终端标志

#define O_TRUNC 01000/*not fcntl*///文件长度截0标志

#define O_APPEND 02000//文件指针置末端标志

#define O_NONBLOCK 04000/*not fcntl*///非阻塞方式打开和操作
文件标志
```

```
#define O_NDELAY O_NONBLOCK
```

```
.....
```

//代码路径: include/fcntl.h: //二进制形式: (可以看出标志的设置有明显的规律)

```
.....
```

```
#define O_ACCMODE 0000 0000 0000 0011
```

```
#define O_RDONLY 0000 0000 0000 0000
```

```
#define O_WRONLY 0000 0000 0000 0001
```

```
#define O_RDWR 0000 0000 0000 0010
```

```
#define O_CREAT 0000 0000 0100 0000/*not fcntl*/
```

```
#define O_EXCL 0000 0000 1000 0000/*not fcntl*/
```

```
#define O_NOCTTY 0000 0001 0000 0000/*not fcntl*/
```

```
#define O_TRUNC 0000 0010 0000 0000/*not fcntl*/
```

```
#define O_APPEND 0000 0100 0000 0000
```

```
#define O_NONBLOCK 0000 1000 0000 0000/*not fcntl*/
```

```
#define O_NDELAY O_NONBLOCK
```

```
.....
```

//代码路径: fs/namei.c:

```

int open_namei (const char * pathname,int fag,int mode,

    struct m_inode ** res_inode) //pathname就是路
径/mnt/user/user1/user2/hello.txt的指针

{

    const char * basename; //basename记录目录项名字前面'/'的地址

    int inr,dev,namelen; //namelen记录名字的长度

    struct m_inode * dir, *inode;

    struct buffer_head * bh;

    struct dir_entry * de; //de用来指向目录项内容

    if ( (flag&O_TRUNC) && ! (flag&O_ACCMODE) ) //如果
文件为只读文件且长度为0

        flag|=O_WRONLY; //将文件设置为只写文件

    mode&=0777&~current->umask;

    mode|=I_REGULAR; //设置该文件为普通文件

    if ( ! (dir=dir_namei (pathname, &namelen, &
basename) ) ) //分析路径、获取枝梢i节点

        return-ENOENT;

    if ( ! namelen ) { /*special case: '/usr/etc*/

        if ( ! (flag& (O_ACCMODE|O_CREAT|O_TRUNC) ) ) {

```

```

    *res_inode=dir;

    return 0;

}

input (dir) ;

return-EISDIR;

}

bh=find_entry (&dir,basename,namelen, &de) ; //通过枝梢i节
点，找到目标文件的目录项

.....

}

```

设置完毕，调用**dir_namei**（）函数分析用户给出的文件路径名，遍历路径所有目录文件i节点，目的是获取最后一个目录文件i节点（枝梢i节点）。

进入dir_namei () 函数后，调用获取i节点的具体工作函数——get_dir () 函数。代码如下：

```
//代码路径： fs/namei.c:

static struct m_inode * dir_namei (const char * pathname,

int * namelen,const char ** name) //pathname就是路
径/mnt/user/user1/user2/hello.txt的指针

{

char c;

const char * basename;

struct m_inode * dir;

if (! (dir=get_dir (pathname) ) ) //分析路径、获取i节点的执行
函数

return NULL;

basename=pathname;

while (c=get_fs_byte (pathname++) ) //遍历结束后， pathname会
指向字符串末端的'\0'

//逐个遍历"/mnt/user/user1/user2/hello.txt"字符串， 每次循环都将一
个字符复制给c
```



```
if (c=='/'){
    basename=pathname; //字符串遍历结束后，basename将指向最后一个 '/'
    *namelen=pathname-basename-1; //计算出"hello.txt"名字的长度
    *name=basename; //得到hello.txt前面 '/' 字符的地址
    return dir;
}
```

`get_dir ()` 函数获取*i*节点的内容，在本书4.1.1节中初步介绍过，获取工作是通过持续不断地“确定目录项、通过目录项获取*i*节点”完成的。

“确定目录项”对应的函数是`find_entry ()`；

“通过目录项获取*i*节点”对应的函数是`iget ()`。

这里详细介绍这两个函数的执行过程。代码如下：

```
//代码路径: fs/namei.c:
```

```
static struct m_inode * get_dir (const char * pathname)
```

```
{
```

```
char c;
```

```
const char * thisname;
```

```
struct m_inode * inode;
```

```
struct buffer_head * bh;
```

```
int namelen,inr,idev;
```

```
struct dir_entry * de;
```

```
if (! current->root||! current->root->i_count) //当前进程的根i节点不存在或引用计数为0
```

```
panic ("No root inode") ;
```

```
if (! current->pwd||! current->pwd->i_count) //当前进程的当前工作目录根i节点不存在或引用计数为0
```

```
panic ("No cwd inode") ;
```

//此处识别出"/mnt/user/user1/user2/hello.txt"这个路径的第一个字符是 '/'

```
if ( (c=get_fs_byte (pathname) ) == '/') {
```

```
inode=current->root;
```

```
pathname++;
```

```
}else if (c)
```

```
inode=current->pwd;
```

```
else
```

```
return NULL; /*empty name is bad*/
```

```
inode->i_count++; //该i节点的引用计数也随之加1
```

```
while (1) { //循环以下过程，直到找到枝梢i节点为止
```

```
thisname=pathname; //thisname会首先指向'm'
```

```
if (! S_ISDIR (inode->i_mode) || ! permission  
(inode,MAY_EXEC) ) {
```

```
input (inode) ;
```

```
return NULL;
```

```
}
```

//每当检索到字符串中的 '/' 字符，或者c为 '\0'，循环都会跳出

```

    for (namelen=0; (c=get_fs_byte (pathname++)) && (c!= '/') ; namelen++)

        /*nothing*/; //注意这个分号

    if (! c)

        return inode;

    if (! (bh=fnd_entry (&inode,thisname,namelen, &de) )) { //通过目录文件的i节点和目录项信息，获取目录项

        iput (inode); //如果在目录文件中没有找到指定的目录项，就释放该目录文件的i节点

        return NULL; //并返回NULL

    }

    inr=de->inode; //从目录项中提取i节点号

    idev=inode->i_dev; //从i节点中获取设备号

    brelse (bh) ;

    iput (inode); //路径中枝梢i节点之前的各个目录文件i节点，使用完毕后就立即释放以免浪费inode_table中的空间

    if (! (inode=iget (idev,inr) )) //获取i节点

        return NULL;

    }

```

```
}
```

`find_entry`（）函数的任务是：先通过目录文件*i*节点，确定目录文件中有多少目录项，之后从目录文件对应的第一个逻辑块开始，不断将该文件的逻辑块从外设读入缓冲区，并从中查找指定目录项，直到找到指定的目录项为止。

代码如下：

```
//代码路径： include/linux/fs.h:
```

```
#define BLOCK_SIZE 1024
```

```
//代码路径： fs/namei.c:
```

```
static struct buffer_head * find_entry (struct m_inode ** dir,
```

```
const char * name,int namelen,struct dir_entry ** res_dir)
```

```
//获取mnt目录项
```

```
{
```

```
int entries;
```

```
int block,i;
```

```
struct buffer_head * bh;
```

```
struct dir_entry * de;
```

```
struct super_block * sb;
```

```
#ifdef NO_TRUNCATE
```

```
    if (namelen > NAME_LEN) //在定义了“NO_TRUNCATE，即不能  
    截断”的前提下，如果文件名超过14字节，就返回NULL
```

```
    return NULL;
```

```
    #else
```

```
    if (namelen > NAME_LEN) //否则就强行截断
```

```
    namelen=NAME_LEN;
```

```
    #endif
```

```
    entries= (*dir) -> i_size/ (sizeof (struct dir_entry) ) ;
```

```
    //根据i节点中i_size文件长度信息，计算目录文件中有多少个目录  
    项
```

```
    *res_dir=NULL;
```

```
    if (! namelen) //检查文件名长度是否为0
```

```

return NULL;

/*check for'..', as we might have to do some"magic"for it*/

if (namelen==2&&get_fs_byte (name) =='.')&&get_fs_byte
(name+1) =='.') {

.....//如果目录项是.., 在这里处理

}

if (! (block= (*dir) ->i_zone[0]) ) //确定目录文件第一个逻辑
块的块号不能是0

return NULL;

if (! (bh=bread ( (*dir) ->i_dev,block) ) ) //将目录项所在逻辑
块读入指定缓冲块

return NULL;

i=0;

de= (struct dir_entry *) bh->b_data; //让de指向缓冲块首地址

while (i<entries) {//在所有目录项中查找mnt目录项

if ( (char *) de>=BLOCK_SIZE+bh->b_data) {

//如果一个缓冲块全部搜索完, 还是没有找到指定的目录项

break (bh) ;

bh=NULL;

```

```

//就将目录文件的下一个逻辑块载入缓冲块，继续找mnt目录项

if ( ! (block=bmap (*dir,i/DIR_ENTRIES_PER_BLOCK) ) ||

    ! (bh=bread ( (*dir) ->i_dev,block) ) ) {

i+=DIR_ENTRIES_PER_BLOCK;

continue;

}

de= (struct dir_entry *) bh->b_data;

}

if (match (namelen,name,de) ) { //目录项匹配确认

*res_dir=de; //如果找到了mnt，就交给*res_dir指针

return bh;

}

de++;

i++;

}

brelse (bh) ;

return NULL; //整个目录文件全都检测完了，确实没有mnt目录
项，返回NULL

```


}

`iget ()` 函数的任务是：根据目录项中提供的i节点号、设备号获取i节点。具体的获取方式是：先在`inode_table[32]`中搜索，如果指定的i节点已在其中，就直接使用；如果找不到，再加载。这样做的理由是：一个文件只能有一个i节点，同一个文件又可能被多个进程同时引用，现在需要获取的文件i节点有可能已经被其他进程载入，如果重复载入i节点，不仅容易引起混乱，而且浪费时间。

此外，如果发现某文件i节点上安装了文件系统，就直接把该文件系统的根i节点载入，这个根i节点将成为在另一个文件系统中继续查找文件的起点。

mnt目录文件的i节点是第一个要获取的i节点，通过5.1节的介绍得知，mnt目录文件i节点上安装了文件系统，这就需要把该文件系统的根i节点载入i节点表。

执行代码如下：

//代码路径： fs/namei.c:

```
struct m_inode * iget (int dev,int nr) //获取mnt目录文件的i节点，
dev和nr分别为指定i节点的设备号和i节点号

{

struct m_inode * inode, *empty;

if (! dev) //如果设备号为空

panic ("iget with dev==0") ;

empty=get_empty_inode () ; //从inode_table[32]中，获取空闲的i
节点表项

inode=inode_table;

//从inode_table[32]中，检测指定的i节点是否已经加载过了，本案
例mnt目录文件i节点就加载过
```

```

while (inode < NR_INODE + inode_table) {

//对比设备号和i节点号是否和指定的i节点相匹配

if (inode->i_dev != dev || inode->i_num != nr) {

inode++;

continue;

}

```

//即便找到了，mnt目录文件i节点此时有可能正在被使用，所以要等待其解锁

```

wait_on_inode (inode) ;

```

if (inode->i_dev != dev || inode->i_num != nr) { //此时该i节点有可能已经

inode = inode_table; //被释放了，因此要重新遍历inode_table[32], 如未被删除

```

continue; //就可以用了，此时的情况是mnt没有被删除

}

```

```

inode->i_count++;

```

if (inode->i_mount) { //如果该i节点上安装了文件系统（本案例中mnt目录文件i节点就是这种情况）

```

int i;

```

```

    for (i=0; i<NR_SUPER; i++) //寻找安装的文件系统所在外设
    (硬盘) , 即hd1设备的超级块

        if (super_block[i].s_imount==inode)

            break;

        if (i>=NR_SUPER) {/i节点上并没有安装文件系统

            printk ("Mounted inode hasn't got sb\n") ;

            if (empty)

                iput (empty) ;

            return inode;

        }

        iput (inode) ;

        dev=super_block[i].s_dev; //通过hd1设备超级块得到设备号

        nr=ROOT_INO; //确定外设根i节点号, ROOT_INO为1

        inode=inode_table; //准备再次遍历外设 (硬盘) 根i节点, 以确定
        其是否也已经加载

        continue;

    }

    if (empty)

```

```

    iput (empty) ;

    return inode;

}

if (! empty) //inode_table[32]中没有空闲项了

return (NULL) ;

//寻找hd1设备根i节点的结果是：没有找到，所以准备加载该i节点

inode=empty;

inode->i_dev=dev;

inode->i_num=nr;

read_inode (inode) ; //读取i节点

return inode; //由于mnt的i节点安装了文件系统，此次获取的是hd1
设备根i节点

}

```

准备工作完成后，通过调用read_inode函数，从外设（此时是硬盘）上读取i节点，载入inode_table[32]中。

执行代码如下：

```
//代码路径： fs/inode.c:
```

```
static void read_inode (struct m_inode * inode) //读取i节点
```

```
{
```

```
    struct super_block * sb;
```

```
    struct buffer_head * bh;
```

```
    int block;
```

```
    lock_inode (inode) ; //为inode_table[32]中指定i节点表项加锁，  
    以免被干扰
```

```
    if ( ! (sb=get_super (inode->i_dev) ) ) //获取i节点所在设备的  
    超级块 (已经加载)
```

```
        panic ("trying to read inode without dev") ;
```

```
    block=2+sb->s_imap_blocks+sb->s_zmap_blocks+
```

```
        (inode->i_num-1) /INODES_PER_BLOCK; //确定i节点在设备  
    上的逻辑块号
```

```
    if ( ! (bh=bread (inode->i_dev,block) ) ) //将i节点所在逻辑块  
    载入指定缓冲块
```

```
        panic ("unable to read i-node block") ;
```

置

```
* (struct d_inode *) inode=//将i节点载入inode_table[32]表的指定位置  
  
    ( (struct d_inode *) bh->b_data)  
  
[ (inode->i_num-1) %INODES_PER_BLOCK];  
  
brelse (bh) ;  
  
unlock_inode (inode) ; //表项操作完毕，解锁  
  
}
```

获取到硬盘上文件系统的根i节点后，get_dir
() 函数将不断地调用find_entry () 函数、iget
() 函数，持续不断获取到user、user1目录文件的i节点，并最终得到user2目录文件的i节点（枝梢i节点）。执行步骤与寻找mnt目录文件i节点一致；区别是，这些目录文件的i节点上没有安装文件系统，执行路径会有所不同。

代码如下：

//代码路径: fs/namei.c:

struct m_inode * iget (int dev,int nr) //获取后续目录文件的i节点,
dev和nr分别为指定i节点的设备号和i节点号

{

struct m_inode * inode, *empty;

if (! dev) //如果设备号为空, 死机

panic ("iget with dev==0") ;

empty=get_empty_inode () ; //从inode_table[32]中, 获取空闲的i
节点表项

inode=inode_table;

while (inode < NR_INODE+inode_table) {

//从inode_table[32]中, 检测指定的i节点是否已经加载过了, 其他
目录文件i节点从未加载过

if (inode->i_dev != dev||inode->i_num != nr) {

//对比的最终结果是循环跳出

inode++;

continue;

}

.....


```

    }

    if (! empty)

        return (NULL) ;

        //寻找其他目录文件i节点的结果是：没有找到，所以准备加载该i
        节点

        inode=empty;

        inode->i_dev=dev;

        inode->i_num=nr;

        read_inode (inode) ; //读取i节点

        return inode; //不断地查找，依次返回的是user、user1、user2目录
        文件i节点，即枝梢i节点

    }

```

执行完毕，返回dir_namei () 函数，将user2目录文件的i节点返回。

执行代码如下：

//代码路径： fs/namei.c:

```

static struct m_inode * dir_namei (const char * pathname,

int * namelen,const char ** name) //pathname就是路
径/mnt/user/user1/user2/hello.txt的指针

{

char c;

const char * basename;

struct m_inode * dir;

if (! (dir=get_dir (pathname) ) ) //获取i节点的执行函数

return NULL;

while (c=get_fs_byte (pathname++) ) //遍历结束后, pathname会
指向字符串末端的'\0'

//逐个遍历"/mnt/user/user1/user2/hello.txt"字符串, 每次循环都将一个
字符复制给c

if (c=='/')

basename=pathname; //字符串遍历结束后, basename将指向最后一
个'/'

*namelen=pathname-basename-1; //计算出"hello.txt"名字的长度

*name=basename; //得到hello.txt前面'/'字符的地址

return dir;

```

```
}
```

最后返回open_namei（）函数中，将user2目录文件的i节点（枝梢i节点）返回并保存。

//代码路径： fs/namei.c:

```
int open_namei (const char * pathname,int fag,int mode,
struct m_inode ** res_inode)
{
.....

if (! (dir=dir_namei (pathname, &namelen, &
basename) )) //通过分析路径得到了枝梢i节点

return-ENOENT;

.....

}
```

`open_namei ()` 函数的任务：通过不断分析路径名最终获取枝梢*i*节点已经完成，下面将通过枝梢*i*节点，确定目标文件hello.txt的*i*节点。

2.获取目标文件*i*节点

获取hello.txt目标文件*i*节点与上一小节中获取枝梢*i*节点的内容基本一致，也通过调用`find_entry ()`、`iget ()` 函数获取目标文件*i*节点，并将其返回。

执行代码如下：

```
//代码路径： fs/namei.c:
```

```
int open_namei (const char * pathname,int flag,int mode,  
struct m_inode ** res_inode)  
{
```

.....

```
if (! (dir=dir_namei (pathname, &namelen, &
basename) ) ) //通过分析路径获得枝梢i节点
```

```
return-ENOENT;
```

```
if (! namelen) { //如果目标文件的名称长度为0
```

```
if (! (flag& (O_ACCMODE|O_CREAT|O_TRUNC) ) ) { //此处
flag检查参见5.2.2节中提供的flag屏蔽码
```

```
*res_inode=dir;
```

```
return 0;
```

```
}
```

```
iput (dir) ;
```

```
return-EISDIR;
```

```
}
```

//通过user2目录文件i节点，以及掌握的关于hello.txt的情况，将hello.txt这一目录项载入缓冲

//块， de指向hello.txt目录项

```
bh=find_entry (& dir,basename,namelen, &de) ;
```

if (! bh) { //hello.txt目录项找到了，缓冲块不可能为空，if中此时不会执行

```

.....

}

inr=de->inode; //得到i节点号

dev=dir->i_dev; //得到hd1设备的设备号

brelse (bh) ;

iput (dir) ; //释放user2目录文件i节点

if (flag&O_EXCL) //此处flag检查参见5.2.2节中提供的flag屏蔽码

return-EEXIST;

if (! (inode=iget (dev,inr) ) ) //获取hello.txt这个文件的i节点

return-EACCES;

if ( (S_ISDIR (inode->i_mode) && (flag&
O_ACCMODE) ) ||//此处flag检查参见5.2.2节中提供的flag屏蔽码

! permission (inode,ACC_MODE (flag) ) ) { //检查用户访问该
文件的许可权限

iput (inode) ;

return-EPERM;

}

inode->i_atime=CURRENT_TIME;

```

```
if (flag & O_TRUNC) //flag检查参见5.2.2节中提供的flag屏蔽码
truncate (inode) ;

*res_inode=inode; //将i节点传递给sys_open

return 0;

}
```

现在已经得到目标文件hello.txt的i节点，下面将这个i节点与file_table[64]挂接。

5.2.3 将文件i节点与file_table[64]挂接

5.2.2节中介绍到，hello.txt文件i节点已经被载入inode_table[32]中。现在要将该i节点与file_table[64]进行挂接，目的是使file_table[64]通过inode_table[32]中hello.txt文件i节点所在表项的指针，找到该i节点。此外，操作系统还对hello.txt文件的属性、引用计数、读写指针偏移等进行了设置。

代码如下：

```
//代码路径： fs/open.c:
```

```
int sys_open (const char * filename,int flag,int mode)
{
.....
}
```


if (S_ISCHR (inode->i_mode)) //hello.txt文件不是字符设备文件，不会执行到这里面

```
if (MAJOR (inode->i_zone[0]) ==4) {  
  
    if (current->leader && current->tty < 0) {  
  
        current->tty=MINOR (inode->i_zone[0]) ;  
  
        tty_table[current->tty].pgrp=current->pgrp;  
  
    }  
  
}else if (MAJOR (inode->i_zone[0]) ==5)  
  
if (current->tty < 0) {  
  
    iput (inode) ;  
  
    current->filp[fd]=NULL;  
  
    f->f_count=0;  
  
    return-EPERM;  
  
}
```

/*Likewise with block-devices: check for floppy_change*/

if (S_ISBLK (inode->i_mode)) //hello.txt文件不是块设备文件，不会执行到这里面

```
check_disk_change (inode->i_zone[0]) ;
```

```
f->f_mode=inode->i_mode; //用该i节点属性，设置文件属性

f->f_flags=flag; //用flag参数，设置文件操作方式

f->f_count=1; //将文件引用计数加1

f->f_inode=inode; //文件与i节点建立关系

f->f_pos=0; //将文件读写指针设置为0

return (fd); //把文件句柄返回用户空间

}
```

到此为止，file_table[64]中的挂接点，一端与当前进程的*filp[20]指针绑定，另一端与inode_table[32]中hello.txt文件的i节点绑定。绑定关系建立后，操作系统把fd返给用户进程。这个fd是挂接点在file_table[64]中的偏移量，即“文件句柄”。进程此后只要把这个fd传递给操作系统，操作系统就可以判断出进程需要操作哪个文件，比如实例1中的

```
int size=read (fd,buffer,sizeof (buffer) ) ;
```

这行程序的目的是要读取hello.txt文件中的内容。实参fd就是hello.txt文件的“标签”。这个参数传入内核后，系统就可以根据fd找到挂接点，并进行操作。

关于读文件操作的详细情况，将在下一节中介绍。

5.3 读文件

读文件就是从用户进程打开的文件中读取数据，读文件由`read`函数完成。

5.3.1 确定数据块在外设中的位置

`read ()` 函数最终映射到`sys_read ()` 系统调用函数去执行。在执行主体内容之前，先对此次操作的可行性进行检查，包括用户进程传递的文件句柄、读取字节数是否在合理范围内，用户进程数据所在的页面能否被写入数据，等等。在这些检查都通过后，开始执行主体内容，即调用`file_read ()` 函数，读取进程指定的文件数据，执行代码如下：

//代码路径: fs/read_write.c:

int sys_read (unsigned int fd,char * buf,int count) //从hello.txt文件中读数据

{//fd是文件句柄, buf是用户空间指针, count是要读取的字节数

struct file * file;

struct m_inode * inode;

if (fd>=NR_OPEN||count<0||! (file=current->filp[fd])) //检查fd、count是否在合理范围内及文件是否已经打开

return-EINVAL;

if (! count) //如果读取字节数为0, 直接返回

return 0;

verify_area (buf,count); //对buf所在页面的属性进行验证, 如果该页面是只读的则复制该页面 (见第6章)

inode=file->f_inode;

if (inode->i_pipe)

return (file->f_mode&1)?read_pipe (inode,buf,count): -EIO;

if (S_ISCHR (inode->i_mode))

return rw_char (READ,inode->i_zone[0], buf,count, &file->f_pos);

```

    if (S_ISBLK (inode->i_mode) )

        return block_read (inode->i_zone[0], &file->f_pos,buf,count) ;

        if (S_ISDIR (inode->i_mode) ||S_ISREG (inode->i_mode) )
        { //分析hello.txt文件i节点属性，得知它为普通文件

            if (count+file->f_pos>inode->i_size)

                count=inode->i_size-file->f_pos;

            if (count<=0)

                return 0;

            return fle_read (inode,file,buf,count) ; //读取进程指定数据

        }

        printk (" (Read) inode->i_mode=%06o\n\r", inode->
i_mode) ; return-EINVAL;

    }

```

在file_read () 中，通过调用bmp () 函数来确定指定的文件数据块在外设上的逻辑块号。执行代码如下：

//代码路径: include/linux/fs.h:

```
#define BLOCK_SIZE 1024
```

//代码路径: fs/fle_dev.c:

```
int file_read (struct m_inode * inode,struct file * filp,char * buf,int  
count) {
```

```
int left,chars,nr;
```

```
struct buffer_head * bh;
```

```
if ( (left=count) <=0)
```

```
return 0;
```

```
while (left) { //每次循环, 最多将一个缓冲块 (1 KB) 的数据复  
制到buf空间内
```

```
if (nr=bmap (inode, (filp->f_pos) /BLOCK_SIZE) ) {
```

```
    //用文件操作指针偏移量除以BLOCK_SIZE (1024), 算出要操作  
    文件中的那一块数据
```

```
    //对于本案例, 此时filp->f_pos为0
```

```
    //根据数据在文件中的数据块号, 确定其在外设上的逻辑块号
```

```
    if ( ! (bh=bread (inode->i_dev,nr) ) ) //从外设上读取数据
```

```
    break;
```

```
    }else
```

```
bh=NULL;
```

```
.....
```

```
}
```

值得注意的是，`bmp ()` 函数调用 `_bmp ()` 函数时，增加了一个参数。

代码如下：

```
//代码路径： fs/inode.c:
```

```
int bmap (struct m_inode * inode,int block)
```

```
{
```

```
    return _bmap (inode,block, 0) ; //最后一个参数是创建标志位。  
置0，表示操作一个已有的块置1，表示创建一个新的块
```

```
}
```

下面先介绍i节点是如何管理文件的。

i节点通过它的i_zone结构来管理文件数据块，具体情景如图5-5～图5-7所示。

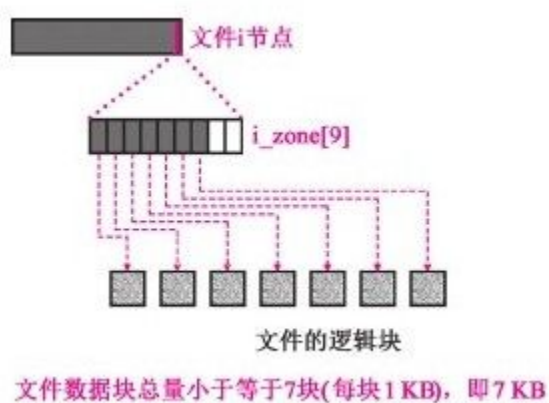


图 5-5 文件数据小于7块时i节点的管理示意图

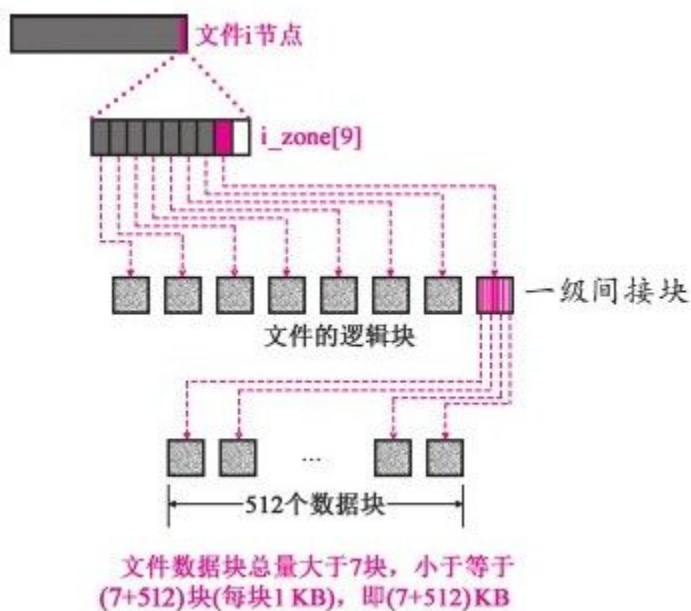


图 5-6 文件数据大于7块、小于 $(7+512)$ 块时
i节点的管理示意图

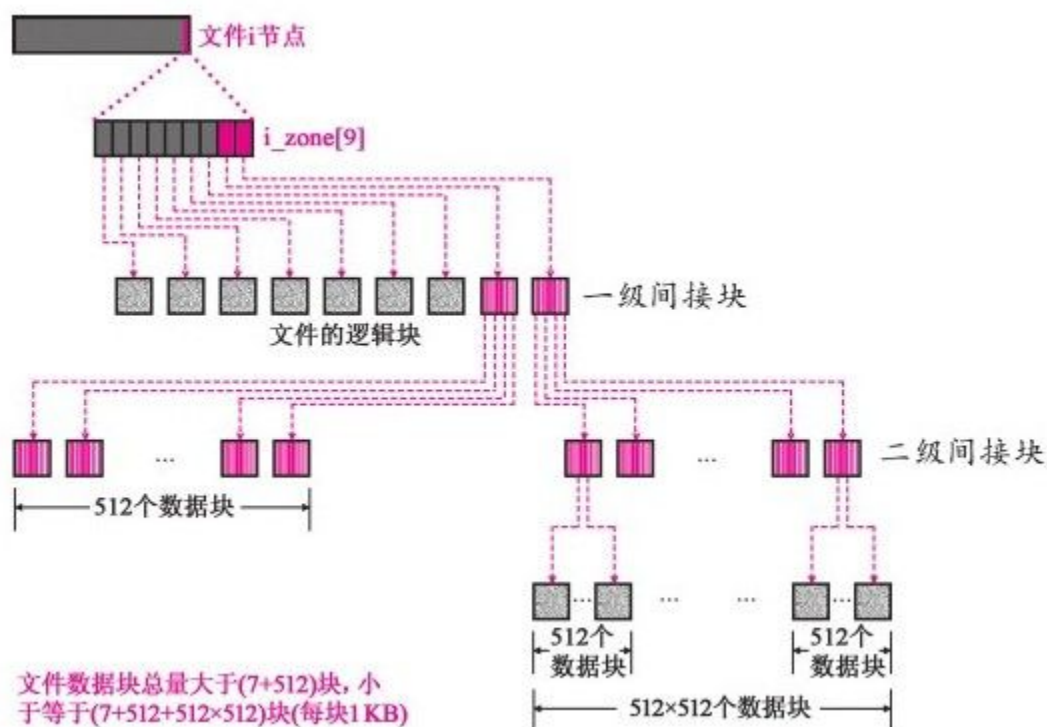


图 5-7 文件数据块大于 $(7+512)$ 、小于minix
允许的极限情况时i节点的管理示意图

i_zone[9]中记录着文件数据块内容的分布情况，但是它毕竟只有9个表项，文件数据块数量如果多于9个就不够用。为此Linux 0.11采取了一种

策略：在数据区中的数据块内继续存储逻辑块的索引值，以此来分级管理数据块，这样就可以增大管理的数据块数量。

当数据总量小于等于7 KB时，i_zone[9]的前7个成员已经足够用了，它们就直接记录该文件的这7个数据块在数据区的“块号”。

当数据量大于7 KB时，就要启动一级间接管理方案。i_zone[9]在其第8个成员记录一个数据块的块号，但这个块里面存储的并不是文件数据内容，而是该文件后续512个数据块在外设中的“逻辑块号”。通过这些块号就可以找到相应的数据块，因为一个数据块的大小为1024字节，而每个块号需要占用两个字节，所以一个数据块能存储512个块号。这样，对数据块进行一级间接管理

时，能够管理的极限应该是 $(7+512)$ 个数据块，即 $(7+512)$ KB。

当数据量大于 $(7+512)$ KB时，就要启动二级间接管理方案，让其第9个成员记录一个数据块的块号。同样，这个块里面存储的并不是文件内容，而是512个数据块在设备中的“逻辑块号”。在这512个数据块中，存储的仍然不是具体的数据内容，而还是索引块。每个块里面又都存储着512个数据块的逻辑块号，这些块号对应的数据块中存储的才是文件的具体数据内容。对数据块进行二级间接管理时，能够管理的极限应该是

$(7+512+512\times 512)$ 个数据块，即

$(7+512+512\times 512)$ KB。

实例1中，此时正在读取hello.txt文件的第一个数据块，属于小于、等于7个逻辑块的情况，执行代码如下：

```
//代码路径： fs/inode.c:

static int_bmap (struct m_inode * inode,int block,int create)

{

    struct buffer_head * bh;

    int i;

    if (block<0) //如果待操作文件数据块号小于0

        panic ("_bmap: block<0") ;

    if (block>=7+512+512*512) //如果待操作文件数据块号大于允许的文件数据块数量最大值

        panic ("_bmap: block>big") ;

    //小于等于7个逻辑块的情况

    if (block<7) { //待操作数据块文件块号小于7

        if (create&&! inode->i_zone[block]) //如果是创建一个新数据块，执行下面代码
```

```

if (inode->i_zone[block]=new_block (inode->i_dev) ) {

inode->i_ctime=CURRENT_TIME;

inode->i_dirt=1;

}

return inode->i_zone[block]; //将i_zone中block项记录的逻辑块号
数值返回

}

//大于7、小于等于（7+512）个逻辑块的情况

block-=7;

if (block<512) { //待操作数据块文件块号小于512，需要一级间
接检索文件块号

if (create && ! inode->i_zone[7]) //如果是创建一个新数据块，
执行下面代码

if (inode->i_zone[7]=new_block (inode->i_dev) ) {

inode->i_dirt=1;

inode->i_ctime=CURRENT_TIME;

}

if (! inode->i_zone[7]) //一级间接块中没有索引号，无法继续查
找，直接返回0

```

```
return 0;
```

```
if ( ! (bh=bread (inode->i_dev,inode->i_zone[7]) ) ) //获取一级间接块
```

```
return 0;
```

```
i= ( (unsigned short *) (bh->b_data) ) [block]; //取该间接块上第block项中的逻辑块号
```

```
if (create&&! i) //如果是创建一个新数据块，执行下面代码
```

```
if (i=new_block (inode->i_dev) ) {
```

```
    ( (unsigned short *) (bh->b_data) ) [block]=i;
```

```
    bh->b_dirt=1;
```

```
}
```

```
    brelse (bh) ;
```

```
    return i;
```

```
}
```

```
//=====大于 (7+512) 、小于 (7+512+512×512) 个逻辑块的情况=====
```

```
    block-=512;
```

```
    if (create&&! inode->i_zone[8]) //如果是创建一个新数据块，执行下面代码
```

```

if (inode->i_zone[8]=new_block (inode->i_dev) ) {

inode->i_dirt=1;

inode->i_ctime=CURRENT_TIME;

}

if (! inode->i_zone[8]) //一级间接块中没有索引号，无法继续查找，直接返回0

return 0;

if (! (bh=bread (inode->i_dev,inode->i_zone[8]) ) ) //获取一级间接块

return 0;

i= ( (unsigned short *) bh->b_data) [block>>9]; //取该间接块上第block/512项中的逻辑块号

if (create&&! i) //如果是创建一个新数据块，执行下面代码

if (i=new_block (inode->i_dev) ) {

( (unsigned short *) (bh->b_data) ) [block>>9]=i;

bh->b_dirt=1;

}

brelse (bh) ;

if (! i)

```



```
return 0;
```

```
if ( ! (bh=bread (inode->i_dev,i) ) ) //获取二级间接块
```

```
return 0;
```

```
    i= ( (unsigned short *) bh->b_data) [block&511]; //取该间接块  
    第二级上第block&511项中的逻辑块号
```

```
    if (create&&! i) //如果是创建一个新数据块，执行下面代码
```

```
    if (i=new_block (inode->i_dev) ) {
```

```
        ( (unsigned short *) (bh->b_data) ) [block&511]=i;
```

```
        bh->b_dirt=1;
```

```
    }
```

```
    brelse (bh) ;
```

```
    return i;
```

```
}
```

5.3.2 将数据块读入缓冲块

调用**bread**（）函数，从硬盘中将**hello.txt**文件的第一个数据块读入指定的缓冲块。情景如图5-8所示。

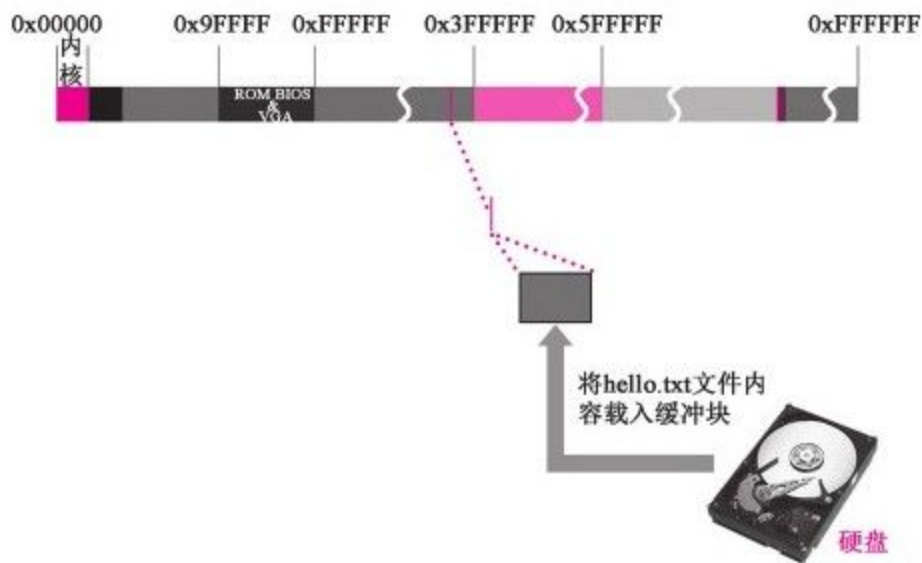


图 5-8 将**hello.txt**文件的数据读入高速缓冲区

执行代码如下：

//代码路径: fs/fle_dev.c:

```
int file_read (struct m_inode * inode,struct file * filp,char * buf,int
count)

{

.....

while (left) { //每次循环最多将一个缓冲块 (1KB) 的数据复制到
buf空间内

    if (nr=bmap (inode, (filp->f_pos) /BLOCK_SIZE) ) { //根据
数据在文件中的数据块号确定其在设备上的逻辑块号

        if ( ! (bh=bread (inode->i_dev,nr) ) ) //从外设上读取数据

            break;

        }else

            bh=NULL;

        .....

    }
```

bread () 函数的详细执行过程, 本书已在
3.3.1节中介绍。

5.3.3 将缓冲块中的数据复制到进程空间

数据块载入缓冲区后，系统要将其从缓冲区复制到指定的用户进程数据空间（*buf）内，执行代码如下：

```
//代码路径： fs/fle_dev.c:

int file_read (struct m_inode * inode,struct file * filp,char * buf,int
count)
{
.....

while (left) {

.....

}else

bh=NULL;

nr=filp->f_pos%BLOCK_SIZE; //以下4行是计算具体要复制多少
字节的数据到用户空间
```

```

chars=MIN (BLOCK_SIZE-nr,left) ;

filp->f_pos+=chars;

left-=chars;

if (bh) {//如果确实从外设上获取到了数据

char * p=nr+bh->b_data;

while (chars-->0) //将chars字节的数据复制到用户指定空间内

put_fs_byte (* (p++) , buf++) ;

brelse (bh) ;

}else{//否则就往指定用户空间复制chars个0

while (chars-->0)

put_fs_byte (0, buf++) ;

}

}

inode->i_atime=CURRENT_TIME;

return (count-left) ? (count-left) : -ERROR;

}

```

把已经读入缓冲块的数据复制到用户执行空间的情景如图5-9所示。

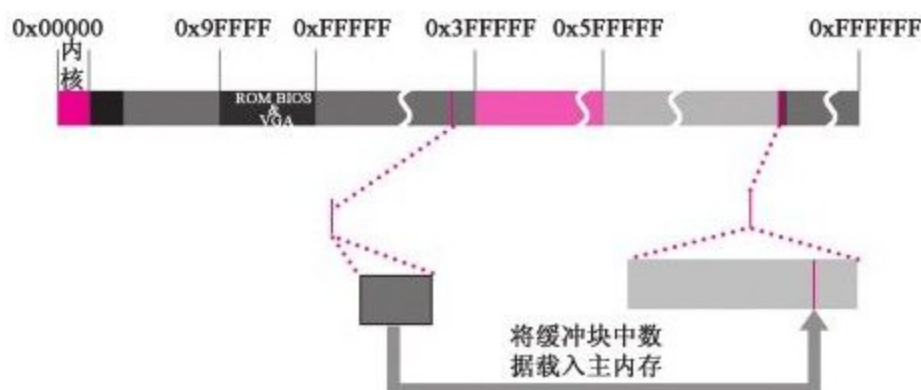


图 5-9 将数据从缓冲块读入主内存

此时只是从hello.txt文件的起始位置读出了一个数据块（1 KB）的数据。通过while不断地循环，将指定数量的数据全部载入用户进程的*buf区域。

读文件操作讲解完毕，下面通过实例2讲解新建文件、写文件操作。

实例2：用户进程在硬盘上新建一个文件，并将内容写入这个文件。

本实例的内容分为两部分：“新建文件”和“写入内容”。实例2对应的进程代码如下：

```
void main ()  
  
{  
  
char str1[]="Hello,world";  
  
//新建文件  
  
int fd=creat ("/mnt/user/user1/user2/hello.txt", 0644) ) ;  
  
//写文件  
  
int size=write (fd,str1, strlen (str1) ) ;  
  
}
```

5.4 新建文件

新建文件就是根据用户进程要求，创建一个文件系统中不存在的文件。新建文件由`creat ()`函数实现。

5.4.1 查找文件

`creat ()` 函数最终映射到`sys_creat ()` 函数中，新建文件和打开文件的代码类似，所以进入`sys_creat ()` 函数后，直接调用`sys_open ()` 函数来新建文件。

执行代码如下：

```
//代码路径： fs/file_dev.c:
```



```
int sys_creat (const char * pathname,int mode) //创建一个新文件

{

    //注意： 创建标志位O_CREAT和独占标志位O_TRUNC全部置位，
    flag参数与5.2.2节中的不同了

    return sys_open (pathname,O_CREAT|O_TRUNC,mode) ;

}
```

通过调用open_namei () 函数来获取hello.txt文件的i节点。

对应代码如下：

```
//代码路径： fs/open.c:

int sys_open (const char * filename,int flag,int mode)

{

    .....

    mode&=0777&~current->umask; //设置该文件模式为用户许可
    使用模式
```

```

for (fd=0; fd<NR_OPEN; fd++)

if (! current->filp[fd])

.....

return-EINVAL;

(current->filp[fd]=f) -> f_count++;

if ( (i=open_namei (filename,flag,mode, &inode) ) < 0) { //获取hello.txt文件i节点

current->filp[fd]=NULL;

.....

}

```

因为是新建文件，此时该文件并不存在，因此open_namei () 函数中的执行情况与5.2.2节中介绍的情况有所区别：调用dir_namei () 函数分析路径名最终获取枝梢i节点后，查找hello.txt目录项，无法找到，bh值将为NULL。

执行代码如下：

```
//代码路径: fs/namei.c:

int open_namei (const char * pathname,int flag,int mode,

struct m_inode ** res_inode)

{

.....

mode&=0777&~current->umask;

mode|=I_REGULAR; //设置该文件为普通文件

if (! (dir=dir_namei (pathname, &namelen, &
basename) )) //分析路径, 获取枝梢i节点

return-ENOENT;

if (! namelen) {/*special case: '/usr/etc*/

if (! (flag& (O_ACCMODE|O_CREAT|O_TRUNC) )) {

*res_inode=dir;

return 0;

}
```

```
input (dir) ;
```

```
return-EISDIR;
```

```
}
```

bh=find_entry (&dir,basename,namelen, &de) ; //通过枝梢i节点, 找到目标文件hello.txt的目录项

```
.....
```

```
}
```

//代码路径: fs/namei.c:

```
static struct buffer_head * find_entry (struct m_inode ** dir,
```

```
const char * name,int namelen,struct dir_entry ** res_dir)
```

```
{
```

```
.....
```

```
i=0;
```

```
de= (struct dir_entry *) bh->b_data; //让de指向缓冲块首地址
```

```
while (i<entries) { //在缓冲块的所有目录项中查找hello.txt目录项
```

if ((char *) de >=BLOCK_SIZE+bh->b_data) { //如果缓冲块中全部搜索完, 还是没有找到hello.txt目录项

```
br else (bh) ;
```

```

bh=NULL;

if ( ! (block=bmap (*dir,i/DIR_ENTRIES_PER_BLOCK) ) ||

    ! (bh=bread ( (*dir) ->i_dev,block) ) ) { //就继续载入目录
项, 继续找

    i+=DIR_ENTRIES_PER_BLOCK;

    continue;

}

de= (struct dir_entry *) bh->b_data;

}

if (match (namelen,name,de) ) { //目录项匹配确认

    *res_dir=de; //如果找到了hello.txt目录项, 就交给*res_dir指针

    return bh;

}

de++;

i++;

}

brelse (bh) ;

```

```
return NULL; //最终没有找到hello.txt目录项
```

```
}
```

5.4.2 新建文件i节点

没有找到hello.txt目录项，并不能确定用户进程的本意就是要新建hello.txt文件（有可能是用户进程把路径名输入错了），所以还要检查新建i节点前，flag中的O_CREAT标志位是否置位。如果确实置位了，就确定用户进程确实是要新建一个文件（5.4.1节中介绍到确实已经置位了）。另外，新建hello.txt文件i节点，将在user2目录文件中写入hello.txt文件对应的新目录项信息，所以还需检查进程对该目录文件是否具备写入权限。之后，再调用new_inode（）函数来新建i节点，并对i节点属性等信息进行设置。

执行代码如下：

//代码路径: fs/namei.c:

```
int open_namei (const char * pathname,int flag,int mode,
```

```
struct m_inode ** res_inode)
```

```
{
```

```
.....
```

```
    if ( ! (dir=dir_namei (pathname, &namelen, &  
    basename) ) ) //分析路径, 获取枝梢i节点
```

```
    return-ENOENT;
```

```
    if ( ! namelen) { /*special case: '/usr/etc*/
```

```
    if ( ! (flag& (O_ACCMODE|O_CREAT|O_TRUNC) ) ) {
```

```
        *res_inode=dir;
```

```
        return 0;
```

```
    }
```

```
    iput (dir) ;
```

```
    return-EISDIR;
```

```
}
```


bh=find_entry (&dir,basename,namelen, &de) ; //通过枝梢i节点, 找到目标文件的目录项

if (! bh) { //没有获取目录项, 缓冲块为空

if (! (flag&O_CREAT)) { //确定用户确实是要新建一个文件

input (dir) ;

return-ENOENT;

}

if (! permission (dir,MAY_WRITE)) { //确定用户是否在user2目录文件中有写入权限

input (dir) ;

return-EACCES;

}

inode=new_inode (dir->i_dev) ; //新建i节点

if (! inode) {

input (dir) ;

return-ENOSPC;

}

inode->i_uid=current->euid; //设置i节点用户id

```

inode->i_mode=mode; //设置i节点访问模式

inode->i_dirt=1; //将i节点已改写标志置1

bh=add_entry (dir,basename,namelen, &de) ; //新建目录项

if (! bh) {

inode->i_nlinks--;

iput (inode) ;

iput (dir) ;

return-ENOSPC;

}

de->inode=inode->i_num;

bh->b_dirt=1;

brelse (bh) ;

iput (dir) ;

*res_inode=inode;

return 0;

}

.....

```

```
}
```

`new_inode`（）函数执行新建i节点的任务分为两部分：

1) 要在i节点位图中，对新建i节点对应的位予以标识。

2) 要将i节点的部分属性信息载入 `inode_table[32]`表中指定的表项。

执行代码如下：

```
//代码路径： fs/bitmap.c:

struct m_inode * new_inode (int dev)

{

    struct m_inode * inode;

    struct super_block * sb;
```

```

struct buffer_head * bh;

int i,j;

if ( ! (inode=get_empty_inode ( ) ) ) //在inode_table[32]中获取
空闲i节点项

return NULL;

if ( ! (sb=get_super (dev) ) ) //获取设备超级块（安装文件系
统时已载入）

panic ("new_inode with unknown device") ;

j=8192; //以下是根据超级块中i节点位图信息， 设置i节点位图

for (i=0; i<8; i++)

if (bh=sb->s_imap[i])

if ( (j=find_first_zero (bh->b_data) ) < 8192)

break;

if ( ! bh||j >=8192||j+i*8192 > sb->s_ninodes) {

input (inode) ;

return NULL;

}

if (set_bit (j,bh->b_data) ) //以上是根据超级块中i节点位图信
息， 设置i节点位图

```

```
panic ("new_inode: bit already set") ;

bh->b_dirt=1; //将i节点位图所在的缓冲块已改写标志置1

//以下对i节点属性进行设置

inode->i_count=1;

inode->i_nlinks=1;

inode->i_dev=dev;

inode->i_uid=current->euid;

inode->i_gid=current->egid;

inode->i_dirt=1;

inode->i_num=j+i*8192;

inode->i_mtime=inode->i_atime=inode->
i_ctime=CURRENT_TIME;

return inode;

}
```

5.4.3 新建文件目录项

hello.txt的目录项要载入user2目录文件中，这里先介绍目录文件的示意图（见图5-10）。

图5-10中的情况1为一个目录文件的初始状态；情况2是将其中的目录项删除（删除的本质就是将目录项中的i节点号清0）；情况3、情况4和情况5都是不断地加载目录项时出现的情况。

调用add_entry（）函数来新建目录项。



图 5-10 目录文件示意图

执行代码如下：

//代码路径： fs/namei.c:

```
int open_namei (const char * pathname,int flag,int mode,struct
m_inode ** res_inode)
```

```
{
```

```
.....
```

```
inode=new_inode (dir->i_dev) ; //新建i节点
```

```

if (! inode) {

input (dir) ;

return-ENOSPC;

}

inode->i_uid=current->euid; //设置i节点用户id

inode->i_mode=mode; //设置i节点访问模式

inode->i_dirt=1; //将i节点使用标志置1

bh=add_entry (dir,basename,namelen, &de) ; //添加目录项

if (! bh) {

inode->i_nlinks--;

input (inode) ;

input (dir) ;

return-ENOSPC;

}

de->inode=inode->i_num; //在目录项中添加i节点号

bh->b_dirt=1;

brelse (bh) ;

```



```
input (dir) ;

*res_inode=inode;

return 0;

}

.....

}
```

`add_entry ()` 函数的任务是：只要在目录文件中寻找到空闲项，就在此位置处加载新目录项；如果确实找不到空闲项，就在外设上创建新的数据块来加载，加载的情景如前面示意图所示。

执行代码如下：

```
//代码路径： fs/namei.c:

static struct buffer_head * add_entry (struct m_inode * dir,
```

const char * name,int namelen,struct dir_entry ** res_dir) //在user2
目录文件中添加目录项

```
{
```

```
int block,i;
```

```
struct buffer_head * bh;
```

```
struct dir_entry * de;
```

```
*res_dir=NULL;
```

```
#ifdef NO_TRUNCATE
```

```
if (namelen > NAME_LEN)
```

```
return NULL;
```

```
#else
```

```
if (namelen > NAME_LEN)
```

```
namelen=NAME_LEN;
```

```
#endif
```

```
if (! namelen)
```

```
return NULL;
```

```
if (! (block=dir->i_zone[0]) ) //确定user2目录文件第一个文件  
块在设备上的逻辑块号 (不能是0)
```

```

return NULL;

    if ( ! (bh=bread (dir->i_dev,block) ) ) //将目录文件的内容载入一个数据块

return NULL;

i=0;

de= (struct dir_entry *) bh->b_data;

while (1) { //在目录文件中搜索空闲目录项

//如果整个数据块中都没有空闲项，就载入下一个数据块继续搜索

//全部载入后仍然没有，就在设备上新建数据块，用以加载新目录项

if ( (char *) de >= BLOCK_SIZE + bh->b_data) {

brelse (bh) ;

bh=NULL;

block=create_block (dir,i/DIR_ENTRIES_PER_BLOCK) ;

if ( ! block)

return NULL;

if ( ! (bh=bread (dir->i_dev,block) ) ) {

i+=DIR_ENTRIES_PER_BLOCK;

```

```
continue;
```

```
}
```

```
de= (struct dir_entry *) bh->b_data;
```

```
}
```

```
//在数据块的末端找到空闲项，就在空闲位置加载目录项
```

```
if (i * sizeof (struct dir_entry) >=dir->i_size) {
```

```
de->inode=0;
```

```
dir->i_size= (i+1) *sizeof (struct dir_entry) ;
```

```
dir->i_dirt=1;
```

```
dir->i_ctime=CURRENT_TIME;
```

```
}
```

```
//在数据块的中间某位置找到空闲项，就在该位置加载目录项
```

```
if (! de->inode) {
```

```
dir->i_mtime=CURRENT_TIME;
```

```
for (i=0; i<NAME_LEN; i++)
```

```
de->name[i]= (i<namelen) ?get_fs_byte (name+i) : 0;
```

```
bh->b_dirt=1;
```

```
*res_dir=de;

return bh;

}

de++;

i++;

}

brelse (bh) ;

return NULL;

}
```

值得注意的是create_block（）函数。调用该函数的执行代码如下：

```
//代码路径： fs/inode.c:

int create_block（struct m_inode * inode,int block）

{
```

//最后一个参数是创建标志位，与本章5.3.1节中不同的是，此时它被置为1，表示有可能要创建新数据块

```
return _bmap (inode,block, 1) ;  
  
}
```

进入_bmp () 函数后，值得关注的代码如下：

//代码路径： fs/inode.c:

```
static int _bmap (struct m_inode * inode,int block,int create)  
{  
  
    struct buffer_head * bh;  
  
    int i;  
  
    if (block<0) //如果待操作文件数据块号小于0  
  
        panic ("_bmap: block<0") ;  
  
    if (block>=7+512+512*512) //如果待操作文件数据块号大于允许的文件数据数量最大值  
  
        panic ("_bmap: block>big") ;
```

//小于等于7个逻辑块的情况

if (block<7) { //待操作数据块文件块号小于7

if (create&&! inode->i_zone[block]) //如果是创建一个新数据块, 执行下面代码

if (inode->i_zone[block]=new_block (inode->i_dev)) {

inode->i_ctime=CURRENT_TIME;

inode->i_dirt=1;

}

return inode->i_zone[block]; //将i_zone中block项记录的逻辑块号数值返回

}

//大于7、小于等于 (7+512) 个逻辑块的情况

block-=7;

if (block<512) { //待操作数据块文件块号小于512, 需要一级间接检索文件块号

if (create&&! inode->i_zone[7]) //如果是创建一个新数据块, 执行下面代码

if (inode->i_zone[7]=new_block (inode->i_dev)) {

inode->i_dirt=1;

```

inode->i_ctime=CURRENT_TIME;

}

if (! inode->i_zone[7]) //一级间接块中没有索引号，无法继续查找，直接返回0

return 0;

if (! (bh=bread (inode->i_dev,inode->i_zone[7]) ) ) //获取一级间接块

return 0;

i= ( (unsigned short *) (bh->b_data) ) [block]; //取该间接块上第block项中的逻辑块

if (create && ! i) //如果是创建一个新数据块，执行下面代码

if (i=new_block (inode->i_dev) ) {

    ( (unsigned short *) (bh->b_data) ) [block]=i;

    bh->b_dirt=1;

}

brelse (bh) ;

return i;

}

//大于 (7+512)、小于 (7+512+512×512) 个逻辑块的情况

```



```
block-=512;
```

if (create && ! inode->i_zone[8]) //如果是创建一个新数据块，
执行下面代码

```
if (inode->i_zone[8]=new_block (inode->i_dev) ) {
```

```
inode->i_dirt=1;
```

```
inode->i_ctime=CURRENT_TIME;
```

```
}
```

if (! inode->i_zone[8]) //一级间接块中没有索引号，无法继续查
找，直接返回0

```
return 0;
```

if (! (bh=bread (inode->i_dev,inode->i_zone[8]))) //获取一
级间接块

```
return 0;
```

i= ((unsigned short *) bh->b_data) [block>>9]; //取该间接块
上第block/512项中的逻辑块号

if (create && ! i) //如果是创建一个新数据块，执行下面代码

```
if (i=new_block (inode->i_dev) ) {
```

```
( (unsigned short *) (bh->b_data) ) [block>>9]=i;
```

```
bh->b_dirt=1;
```

```

    }

    brelse (bh) ;

    if (! i)

    return 0;

    if (! (bh=bread (inode->i_dev,i) ) ) //获取二级间接块

    return 0;

    i= ( (unsigned short *) bh->b_data) [block&511]; //取该间接块
    第二级上第block&511项中的逻辑块号

    if (create&&! i) //如果是创建一个新数据块，执行下面代码

    if (i=new_block (inode->i_dev) ) {

        ( (unsigned short *) (bh->b_data) ) [block&511]=i;

        bh->b_dirt=1;

    }

    brelse (bh) ;

    return i;

}

```

`create`标志置位，不等于就要创建一个新数据块，必须确保文件的下一个文件块不存在，即！`inode->i_zone[.....]`或！`i`成立，才能创建新数据块。比如本实例中加载目录项的内容，一个数据块中没有发现空闲项，很可能下一个数据块中就有，如果强行分配新数据块，就会把已有的块覆盖掉，导致目录文件管理混乱。

新建数据块的工作在`new_block()`函数中执行，将在本章5.5节中详细介绍。

新建目录项的情景如图5-11所示。

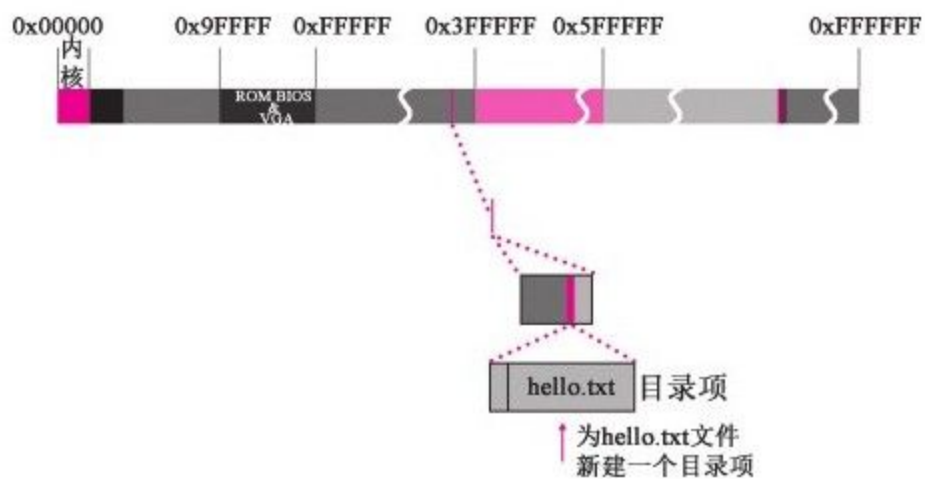


图 5-11 查找空目录项并添加目录数据

5.5 写文件

操作系统对写文件操作的规定是：进程空间的数据先要写入缓冲区中，然后操作系统在适当的条件下，将缓冲区中的数据同步到外设上。而且，操作系统只能以数据块（1 KB）为单位，将缓冲区中的缓冲块（1 KB）的数据同步到外设上。这就需要在同步之前，缓冲块与外设上要写入的逻辑块进行一对一绑定，确定外设上的写入位置，以此保证用户空间写入缓冲块的数据，能够准确地同步到指定逻辑块中。

首先介绍如何确定绑定关系。

5.5.1 确定文件的写入位置

`write ()` 函数最终映射到 `sys_write ()` 函数中去执行。该函数先对参数的合理性进行检查,之后调用 `file_write ()` 函数写文件。

执行代码如下:

```
//代码路径: fs/read_write.c:

int sys_write (unsigned int fd,char * buf,int count) //写文件

{

    struct file * file;

    struct m_inode * inode;

    if (fd >= NR_OPEN || count < 0 || ! (file=current-> filp[fd]) ) //fd 、
count是否在合理范围内及文件是否已经打开

        return-EINVAL;

    if (! count) //如果写入字节数为0, 直接返回

        return 0;

    inode=file-> f_inode;
```

```

    if (inode->i_pipe)

        return (file->f_mode&2) ?write_pipe (inode,buf,count) : -
EIO;

    if (S_ISCHR (inode->i_mode) )

        return rw_char (WRITE,inode->i_zone[0], buf,count, &file->
f_pos) ;

    if (S_ISBLK (inode->i_mode) )

        return block_write (inode->i_zone[0], &file->
f_pos,buf,count) ;

    if (S_ISREG (inode->i_mode) ) //确定待写入文件是普通文件

        return file_write (inode,file,buf,count) ; //写文件

    printk (" (Write) inode->i_mode=%06o\n\r", inode->
i_mode) ;

    return-EINVAL;

}

```

用户进程传递的flags参数，决定了文件的数据写入位置。因此进入file_write () 函数后，先检查f_flags标志位来确定写入位置，之后，调用

`create_block ()` 函数，创建一个与该文件位置对应的外设逻辑块，并返回逻辑块号。

执行代码如下：

```
//代码路径: fs/fle_dev.c:

int file_write (struct m_inode * inode,struct file * filp,char * buf,int
count)

{

    off_t pos;

    int block,c;

    struct buffer_head * bh;

    char * p;

    int i=0;

    /*

    *ok,append may not work when many processes are writing at the
same time

    *but so what.That way leads to madness anyway.
```



```

*/

if (filp->f_flags&O_APPEND) //如果设置了文件尾部加写标志

pos=inode->i_size; //pos移至文件尾部

else

pos=filp->f_pos; //直接从文件指针f_pos当前指向的位置处开始
写入数据（本案例是这种情况，f_pos为0）

while (i<count) {

    if (! (block=create_block (inode,pos/BLOCK_SIZE) ) ) //创建
逻辑块，并返回块号

        break;

    if (! (bh=bread (inode->i_dev,block) ) ) //申请缓冲块（不需
要读出来）

        break;

    .....

}

```

创建一个新的数据块，并使之与i节点中指定的i_zone[9]对应。

执行代码如下：

```
//代码路径： fs/inode.c:
```

```
int create_block (struct m_inode * inode,int block)
```

```
{
```

```
    return _bmap (inode,block, 1) ; //最后一个参数是创建标志位，  
置1，表示创建一个新的块
```

```
}
```

```
//代码路径： fs/inode.c:
```

```
static int _bmap (struct m_inode * inode,int block,int create)
```

```
{
```

```
    struct buffer_head * bh;
```

```
    int i;
```

```
    if (block < 0) //如果待操作文件数据块号小于0
```

```
        panic ("_bmap: block < 0") ;
```

```
    if (block >= 7+512+512*512) //如果待操作文件数据块号大于允  
许的文件数据块数量最大值
```

```
        panic ("_bmap: block > big") ;
```

```

    if (block < 7) { //待操作数据块文件块号block小于7，本案例中
block为0

        if (create && ! inode->i_zone[block]) //创建一个新数据块

            if (inode->i_zone[block]=new_block (inode->i_dev) ) { //将新建
数据块和//i节点中的i_zone[9]对应

                inode->i_ctime=CURRENT_TIME;

                inode->i_dirt=1;

            }

            return inode->i_zone[block]; //将i_zone[9]中block项记录的逻辑块
号数值返回

        }

        .....

    }

```

具体的创建工作是在new_block () 函数中进行的，内容包括两部分：

- 1) 将新建数据块对应的逻辑块位图置1。

2) 在缓冲区中为新建的数据块申请缓冲块,用以承载写入的内容。

执行代码如下:

```
//代码路径: fs/bitmap.c:

int new_block (int dev) //创建一个新数据块

{

    struct buffer_head * bh;

    struct super_block * sb;

    int i,j;

    if ( ! (sb=get_super (dev) ) ) //获取设备的超级块 (安装文件
系统时已载入)

        panic ("trying to get new block from nonexistant device" ) ;

    j=8192;

    //以下是根据超级块中逻辑块位图信息, 对新数据块的逻辑块位图
    进行设置

    for (i=0; i<8; i++)
```

```

if (bh=sb->s_zmap[i])

if ( (j=find_first_zero (bh->b_data) ) < 8192)

break;

if (i>=8|| ! bh||j>=8192)

return 0;

if (set_bit (j,bh->b_data) ) //以上是根据超级块中逻辑块位图信息,
对新数据块的逻辑块位图进行设置

panic ("new_block: bit already set" ) ;

bh->b_dirt=1; //将逻辑块位图所在的缓冲块使用标记置1 (已改
写)

j+=i*8192+sb->s_firstdatazone-1; //确定数据块逻辑块号

if (j>=sb->s_nzones)

return 0;

if ( ! (bh=getblk (dev,j) ) ) //在缓冲区中, 为新的数据块申请
一个空闲缓冲块

panic ("new_block: cannot get block" ) ;

if (bh->b_count!=1)

panic ("new block: count is !=1" ) ;

clear_block (bh->b_data) ; //将该逻辑块中数据清零

```

```
bh->b_uptodate=1; //已更新标志被置1
```

```
bh->b_dirt=1; //已改写标志被置1
```

```
brelse (bh) ;
```

```
return j;
```

```
}
```

5.5.2 申请缓冲块

调用**bread**（）函数，由于**new_block**（）函数创建的是新缓冲块，所以就无须从外设上载入逻辑块了，执行代码如下：

//代码路径： fs/file_dev.c:

```
int file_write (struct m_inode * inode,struct file * filp,char * buf,int count)
```

```
{
```

```
.....
```

```
while (i < count) {
```

```
    if (! (block=create_block (inode,pos/BLOCK_SIZE) ) ) //创建新的数据块
```

```
        break;
```

```
    if (! (bh=bread (inode->i_dev,block) ) ) //载入缓冲块
```

```
        break;
```

```
c=pos%BLOCK_SIZE;
```

```
.....
```

```
}
```

```
//代码路径: fs/buffer.c:
```

```
struct buffer_head * bread (int dev,int block)
```

```
{
```

```
struct buffer_head * bh;
```

```
    if (! (bh=getblk (dev,block) )) //数据块已与哈希表挂接, 不需要从外设上读取
```

```
panic ("bread: getblk returned NULL\n") ;
```

```
    if (bh->b_uptodate) //b_uptodate在new_block () 中被置1, 直接返回
```

```
return bh;
```

```
ll_rw_block (READ,bh) ;
```

```
wait_on_buffer (bh) ;
```

```
if (bh->b_uptodate)
```

```
return bh;
```

```
brelse (bh) ;
```



```
return NULL;
```

```
}
```

5.5.3 将指定的数据从进程空间复制到缓冲块

将数据复制到指定缓冲块的情景如图5-12所示。

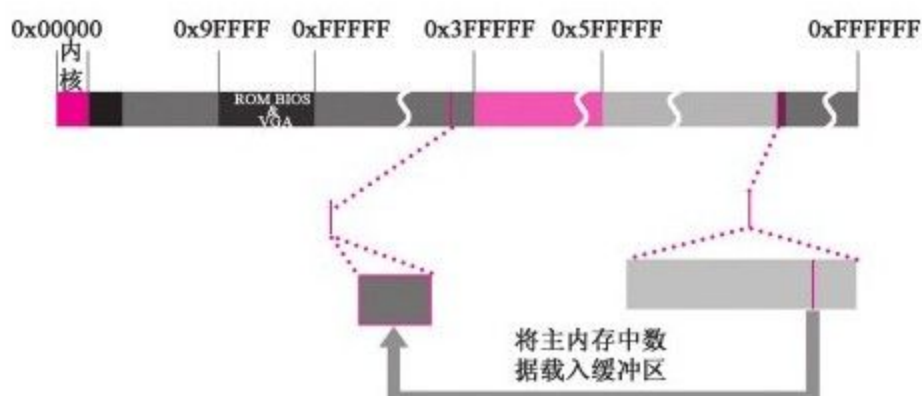


图 5-12 将数据复制到指定缓冲块

即将写入的字符为“Hello,world”，一个缓冲块足以承载了，因此while循环只进行一次。执行代码如下：

//代码路径: fs/file_dev.c:

```
int file_write (struct m_inode * inode,struct file * filp,char * buf,int  
count)
```

```
{
```

```
.....
```

```
if (! (bh=bread (inode->i_dev,block) ) ) //申请缓冲块 (不需  
要读出来)
```

```
break;
```

```
c=pos%BLOCK_SIZE; //以下代码计算向缓冲块中写入字节数
```

```
p=c+bh->b_data;
```

```
bh->b_dirt=1;
```

```
c=BLOCK_SIZE-c;
```

```
if (c>count-i) c=count-i;
```

```
pos+=c;
```

```
if (pos>inode->i_size) {
```

```
inode->i_size=pos;
```

```
inode->i_dirt=1;
```

```
}
```

```

i+=c;

while (c-->0)

* (p++) =get_fs_byte (buf++) ; //将数据写入指定的缓冲块

brelse (bh) ;

}

inode->i_mtime=CURRENT_TIME;

if ( ! (filp->f_flags&O_APPEND) ) {

filp->f_pos=pos;

inode->i_ctime=CURRENT_TIME;

}

return (i?i: -1) ;

}

```

此时，用户进程指定的数据，只是写入缓冲区中，并未写入硬盘。下面介绍数据从缓冲区同步到硬盘的方式。

5.5.4 数据同步到外设的两种方法

数据从缓冲区同步到硬盘有两种方法。一种是updata定期同步；另一种是因缓冲区使用达到极限，操作系统强行同步。

第一种方法：

本书4.4.1节中介绍到，shell进程在第一次执行时，启动了一个updata进程。这个进程常驻于内存，功能就是将缓冲区中的数据同步到外设上。

该进程会调用pause（）函数，这个函数会映射到sys_pause（）函数中，使该进程被设置为可中断等待状态。每隔一段时间，操作系统就将

update进程唤醒。它执行后，调用sync（）函数，将缓冲区中的数据同步到外设上。

sync（）函数最终映射到sys_sync（）系统调用函数去执行。为了保证文件内容同步的完整性，需要将文件i节点位图、文件i节点、文件数据块、数据块对应的逻辑块位图，全都同步到外设。sys_sync（）函数先将改动过的文件i节点写入缓冲区（其余内容已经在缓冲区中了），之后，遍历整个缓冲区，只要发现其中缓冲块内容被改动过（b_dirt被置1），就全部同步到外设上。

执行代码如下：

```
//代码路径： fs/buffer.c:
```

```
int sys_sync（void）
```

```

{

int i;

struct buffer_head * bh;

sync_inodes () ; //将i节点写入缓冲区

bh=start_buffer;

for (i=0; i<NR_BUFFERS; i++, bh++) { //遍历整个缓冲区

    wait_on_buffer (bh) ; //如果哪个缓冲块正在使用，就等待这个
    缓冲块解锁

    if (bh->b_dirt) //只要这个缓冲块中的内容被改写

        ll_rw_block (WRITE,bh) ; //将该缓冲块的内容同步到外设中

    }

return 0;

}

```

同步i节点的任务是由sync_inode函数完成的。

执行代码如下：

```
//代码路径: fs/inode.c:
```

```
void sync_inodes (void)
```

```
{
```

```
int i;
```

```
struct m_inode * inode;
```

```
inode=0+inode_table;
```

```
for (i=0; i<NR_INODE; i++, inode++) { //遍历所有i节点
```

```
    wait_on_inode (inode) ; //如果遍历到的i节点正在使用就等待该i  
节点解锁
```

```
    if (inode->i_dirt && ! inode->i_pipe) //如果i节点内容已经改动  
过, 而且不是管道文件的i节点
```

```
        write_inode (inode) ; //将i节点同步到缓冲区
```

```
}
```

```
}
```

```
//代码路径: fs/inode.c:
```

```
static void write_inode (struct m_inode * inode)
```



```

{

struct super_block * sb;

struct buffer_head * bh;

int block;

lock_inode (inode) ; //先将i节点加锁， 以免被干扰

if ( ! inode->i_dirt || ! inode->i_dev ) {

unlock_inode (inode) ;

return;

}

if ( ! (sb=get_super (inode->i_dev) ) ) //获取外设超级块

panic ("trying to write inode without device" ) ;

block=2+sb->s_imap_blocks+sb->s_zmap_blocks+

    (inode->i_num-1) /INODES_PER_BLOCK; //确定i节点位图在
    外设上的逻辑块号

    if ( ! (bh=bread (inode->i_dev,block) ) ) //将i节点所在逻辑块
    载入缓冲区

panic ("unable to read i-node block" ) ;

    ( (struct d_inode *) bh->b_data) //将i节点同步到缓冲区

```

```
[ (inode->i_num-1) %INODES_PER_BLOCK]=  
* (struct d_inode *) inode;  
  
bh->b_dirt=1; //将缓冲块的b_dirt置1  
  
inode->i_dirt=0; //将i节点的i_dirt置0  
  
brelse (bh) ;  
  
unlock_inode (inode) ; //i节点解锁  
  
}
```

同步工作完成后，update进程将被挂起，下一次被唤醒后，继续同步缓冲区。

第二种方法：

实例2的场景比较简单，它写入缓冲区的数据比较少，我们不妨对其稍加改动，来看下面源代码：

```
void main ()  
  
{  
  
char str1[]="Hello,world";  
  
int i;  
  
//新建文件  
  
int fd=creat ("/mnt/user/user1/user2/hello.txt", 0644) ) ;  
  
//写文件  
  
for (i=0; i<1000000; i++)  
  
{  
  
int size=write (fd,str1, strlen (str1) ) ;  
  
}  
  
}
```

要写入的数据将达到**10 MB**以上，而缓冲区，肯定不可能超过**10 MB**，因此，当前进程要写入数据的话，很可能在update进程被唤醒之

前，就已经将缓冲区写满，若继续写入，就需要强行将缓冲区中的数据同步到硬盘，为续写腾出空间。

此任务是由`getblk`（）函数完成的。本书3.3.1节中已经对此函数进行过介绍，当在缓冲区中找到的空闲块都已经无法继续写入信息（`b_dirt`都是1）时，就说明需要腾空间了。

执行代码如下：

```
//代码路径： fs/buffer.c:

struct buffer_head * getblk (int dev,int block)

{

    struct buffer_head * tmp, *bh;

    repeat:

    if (bh=get_hash_table (dev,block) )
```

```

return bh;

tmp=free_list;

do{

if (tmp->b_count)

continue;

if (! bh||BADNESS (tmp) <BADNESS (bh) ) {

bh=tmp;

if (! BADNESS (tmp) )

break;

}

/*and repeat until we find something good*/

}while ( (tmp=tmp->b_next_free) !=free_list) ; //找到空闲的
缓冲块 (不等价于b_dirt是0)

if (! bh) {

sleep _on (&buffer_wait) ;

goto repeat;

}

```

```
wait_on_buffer (bh) ;
```

```
if (bh->b_count)
```

```
goto repeat;
```

while (bh->b_dirt) { //虽然找到空闲缓冲块, 但b_dirt仍是1, 说明缓冲区中已无可用的缓冲块了, 需要同步腾空

```
sync_dev (bh->b_dev) ; //同步数据
```

```
wait_on_buffer (bh) ;
```

```
if (bh->b_count)
```

```
goto repeat;
```

```
}
```

/*NOTE! While we slept waiting for this block,somebody else might*/

```
/*already have added"this"block to the cache.check it*/
```

```
if (find_buffer (dev,block) )
```

```
goto repeat;
```

/*OK,FINALLY we know that this buffer is the only one of it's kind,
*/

/*and that it's unused (b_count=0) , unlocked (b_lock=0) , and
clean*/

```
bh->b_count=1;

bh->b_dirt=0;

bh->b_uptodate=0;

remove_from_queues (bh) ;

bh->b_dev=dev;

bh->b_blocknr=block;

insert_into_queues (bh) ;

return bh;

}
```

以上就是数据同步的两种策略。

值得注意的是，5.5.3节中p指向的数据块是新申请的，之前没有内容，从指定数据块的起始位置开始写入数据，不会影响已有数据；如果hello.txt不是一个新建文件，而是已有文件，则指针p无论往哪个数据块写入数据，都会把这个数据

块中写入点后面已有的数据覆盖掉（除非是在尾端加写）。

这意味着用户只能在文件尾端“加写”数据；如果想在文件中间“改写”数据，单纯依靠 `sys_write()` 函数无法做到。那么面对“改写”数据这种更为复杂的文件写入情况，操作系统又该如何处理呢？下面将详细讲解。

5.6 修改文件

修改文件的本质就是可以在文件的任意位置插入数据、删除数据，且不影响文件已有数据。

此问题的处理方案是：将`sys_read()`、`sys_write()`以及`sys_lseek()`几个函数组合使用。

`sys_read()`和`sys_write()`函数已在5.3节和5.5节中介绍。这里先介绍`sys_lseek()`函数，之后再介绍如何组合使用它们。

5.6.1 重定位文件的当前操作指针

用户进程调用`lseek()`函数，将文件的当前操作指针`f_pos`进行重新定位，它最终映射到`sys_lseek()`函数去执行。

执行代码如下：

```
//代码路径： include/Unistd.h:
```

```
#define SEEK_SET 0//表明从文件起始处开始偏移
```

```
#define SEEK_CUR 1//表明从文件的当前读写位置处开始偏移
```

```
#define SEEK_END 2//表明从文件的尾端开始偏移
```

```
//代码路径： fs/read_write.c:
```

```
int sys_lseek (unsigned int fd,off_t offset,int origin) //调整文件操作  
指针， offset是f_pos向文件尾端偏移字节数
```

```
{
```

```
struct file * file;
```

```
int tmp;
```

```
if (fd >= NR_OPEN || ! (file=current->filp[fd]) || ! (file->  
f_inode)
```

```
|| ! IS_SEEKABLE (MAJOR (file->f_inode->i_dev) ) )
```

```
return-EBADF;
```

```
if (file->f_inode->i_pipe)
```

```
return-ESPIPE;
```

```
switch (origin) {

case 0:

//以文件起始处作为起点， 设置file->f_pos

if (offset<0) return-EINVAL;

file->f_pos=offset;

break;

case 1:

//将file->f_pos设置为文件当前操作位置

if (file->f_pos+offset<0) return-EINVAL;

file->f_pos+=offset;

break;

case 2:

//以文件末尾为起点， 设置file->f_pos

if ( (tmp=file->f_inode->i_size+offset) <0)

return-EINVAL;

file->f_pos=tmp;

break;
```

default:

return-EINVAL;

}

return file->f_pos;

}

5.6.2 修改文件

现在，假设hello.txt是硬盘上已有的一个文件，而且内容为“hello,world”，这里介绍通过sys_read（）函数、sys_write（）函数和sys_lseek（）函数联合使用，把数据插入hello.txt文件中。

进程的程序代码如下：

```
#include <fcntl.h>

#include <stdio.h>

#include <string.h>

#define LOCATION 6

int main (char argc,char ** argv)

{

char str1[]="Linux";
```

```
char str2[1024];

int fd,size;

memset (str2, 0, sizeof (str2) ) ;

fd=open ("hello.txt", O_RDWR, 0644) ;

lseek (fd,LOCATION,SEEK_SET) ;

strcpy (str2, str1) ;

size=read (fd,str2+5, 6) ;

lseek (fd,LOCATION,SEEK_SET) ;

size=write (fd,str2, strlen (str2) ) ;

close (fd) ;

return 0;

}
```

这段程序的意思是将“Linux”这个字符串插入hello.txt文件中了，最终hello.txt文件中的内容应该是“hello,Linuxworld”。

```
fd=open ("hello.txt", O_RDWR, 0644) ;
```

`open ()` 函数将对应`sys_open ()` 函数，打开即将操作的文件。

```
lseek (fd,LOCATION,SEEK_SET) ;
```

`lseek ()` 函数将对应`sys_lseek ()` 函数；参数中选择了`SEEK_SET`，表明要将文件的当前操作指针从文件的起始位置向文件尾端偏移6字节。

```
strcpy (str2, str1) ;
```

这一行是将“Linux”这个字符串复制到`str2[1024]`这个数组的起始位置处。

```
size=read (fd,str2+5, 6) ;
```

`read ()` 函数将对应 `sys_read ()` 函数，读取 `hello.txt` 这个文件的内容；实参“`str2+5`”表示要把从 `hello.txt` 这个文件读出来的内容复制到 `str2` 这个数组的第6个元素之后，相当于与 `Linux` 这个字符串进行了拼接；实参“`6`”表明要将该文件的6个字符读出，前面 `lseek`

`(fd, LOCATION, SEEK_SET)`；已经将文件的当前操作指针从文件的起始位置向文件尾端偏移了6字节，所以此次读出的内容就应该是 `world`，最后拼接的结果是 `Linuxworld`。 `lseek`

`(fd, LOCATION, SEEK_SET)`；

这行的效果和前面调用的效果一样，都是要将文件的当前操作指针从文件的起始位置，向文件尾端偏移6字节，以此确定文件的写入位置。


```
size=write (fd,str2, strlen (str2) ) ;
```

`write ()` 函数将对应`sys_write ()` 函数。现在要将`str2`这个数组中的“Linuxworld”字符串写入`hello.txt`文件中，而且写入位置就是从文件的起始点向尾端偏移6字节的位置，最终的写入结果就是“hello,Linuxworld”。

实例3：关闭此文件，之后将其从文件系统中删除

我们不妨将5.6节中的程序改写，代码如下：

```
#include <fcntl.h>

#include <stdio.h>

#include <string.h>

#define LOCATION 6
```

```
int main (char argc,char ** argv)

{

char str1[]="Linux";

char str2[1024];

int fd,size;

memset (str2, 0, sizeof (str2) ) ;

fd=open ("/mnt/user/user1/user2/hello.txt", O_RDWR, 0644) ;

lseek (fd,LOCATION,SEEK_SET) ;

strcpy (str2, str1) ;

size=read (fd,str2+5, 6) ;

lseek (fd,LOCATION,SEEK_SET) ;

size=write (fd,str2, strlen (str2) ) ;

//关闭文件

close (fd) ;

//删除文件

unlink ("/mnt/user/user1/user2/hello.txt") ;

return 0;
```

}

5.7 关闭文件

关闭文件对应的是打开文件，是在`close ()`函数中完成的。

5.7.1 当前进程的`filp`与`file_table[64]`脱钩

`close ()` 函数最终映射到`sys_close ()` 系统调用函数去执行。将当前进程的`task_struct`中的`filp[20]`与`file_table[64]`解除关系，情景如图5-13所示。

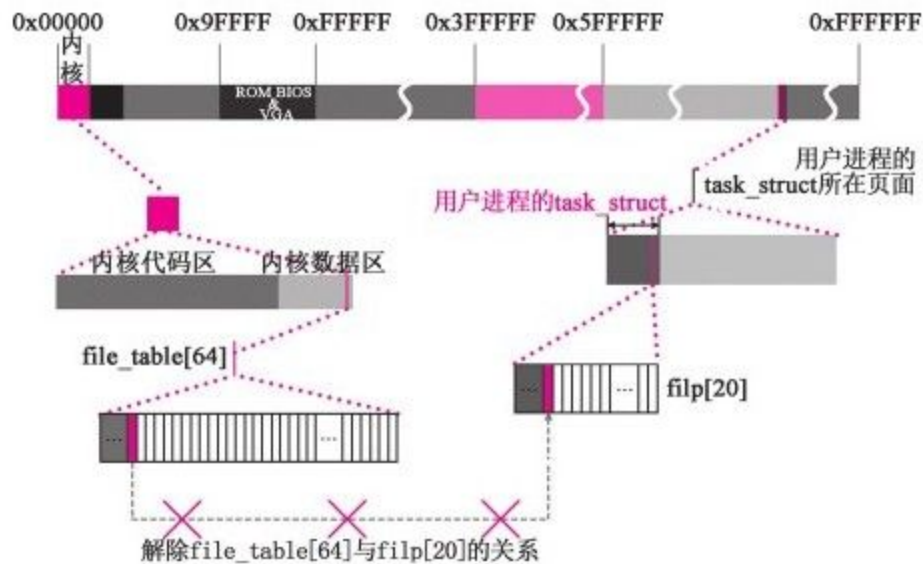


图 5-13 当前进程的filp[20]与file_table[64]脱钩

执行代码如下：

//代码路径： fs/open.c:

```
int sys_close (unsigned int fd)
```

```
{
```

```
struct file * filp;
```

```
if (fd >= NR_OPEN)
```

```
return-EINVAL;
```

```
current->close_on_exec&=~ (1 << fd) ;
```

```
if (! (filp=current-> filp[fd]) )  
  
return-EINVAL;  
  
current-> filp[fd]=NULL; //将当前进程filp[20]中的fd项置空  
  
if (filp-> f_count==0)  
  
panic ("Close: file count is 0") ;  
  
if (--filp-> f_count) //将file_table[64]中文件句柄引用计数递减  
  
return (0) ;  
  
iput (filp-> f_inode) ; //将i节点与file_table[64]脱钩  
  
return (0) ;  
  
}
```

值得注意的是，file_table[64]用来管理操作系统中所有正在操作的文件，此时其他进程可能正在操作hello.txt文件，而且共用一本账（这种情况我们在5.2.1节中讲解close_on_exec字段时已经介绍），因此，filp-> f_count只被减少了引用计

数，而没有被简单地清空。当然，实例3中，没有进程操作该文件，`filp->f_count`将递减为0，`file_table[64]`中的这个表项变成了空闲项。

5.7.2 文件i节点被释放

文件i节点被释放的过程是：先要对i节点的各类属性进行检查，对于实例3，将会检查到i节点的内容已被改变，因此，要先将i节点同步到指定缓冲块中；然后，递减i节点的i_count，使i节点的引用计数变为0，这个i节点在inode_table[32]中的表项成为空闲项。

执行代码如下：

//代码路径：fs/open.c:

```
void iput (struct m_inode * inode) //释放文件i节点
{
    if (! inode)
        return;
```


`wait_on_inode (inode) ;` //i节点可能正在被使用，所以要等待i节点解锁

`if (! inode->i_count) //如果即将释放的i节点引用计数为0`

`panic ("input: trying to free free inode") ;`

`if (inode->i_pipe) {`//如果该i节点是管道文件的i节点

`wake_up (&inode->i_wait) ;`

`if (--inode->i_count)`

`return;`

`free_page (inode->i_size) ;`

`inode->i_count=0;`

`inode->i_dirt=0;`

`inode->i_pipe=0;`

`return;`

`}`

`if (! inode->i_dev) {`//如果i节点所在外设的设备号为0

`inode->i_count--;` //其引用计数递减

`return;`

```

    }

    if (S_ISBLK (inode->i_mode) ) { //如果i节点是块设备文件的i
节点
        sync_dev (inode->i_zone[0]) ; //将其同步到外设上

        wait _on_inode (inode) ;

    }

repeat:

    if (inode->i_count > 1) { //i节点的引用计数大于1

        inode->i_count--; //递减其引用计数

        return;

    }

    if (! inode->i_nlinks) { //如果i节点的链接数为0

        truncate (inode) ; //释放该i节点对应的所有逻辑块

        free_inode (inode) ; //释放该i节点

        return;

    }

    if (inode->i_dirt) { //对于本案例，i节点内容已经改变，同步该i
节点内容到外设

```

```
write_inode (inode) ; /*we can sleep-so do again*/
```

```
wait_on_inode (inode) ;
```

```
goto repeat;
```

```
}
```

```
inode->i_count--; //i节点引用计数递减
```

```
return;
```

```
}
```

5.8 删除文件

删除文件对应的是新建文件。删除文件与5.7节中关闭文件有所不同：关闭文件只是解除当前进程与hello.txt文件在file_table[64]中指定挂接点的关系，而删除操作的效果表现为所有进程都无法访问到hello.txt这个文件。

小贴士

在Linux 0.11中，允许利用系统调用函数sys_link将“/mnt/user/zhang/chengxu.c”的路径名指向路径名为“/mnt/user/hello.txt”的文件，类似Windows下的快捷方式。这样可以允许不同用户建立自己熟悉的路径名和文件名来访问他想访问的文件，而不是必须要记住最原始的路径名。在i

节点中，利用*i_nlinks*字段标识有多少个路径名（目录项）链接到一个文件。每建立一个这样的链接，*i_nlinks*就增加1。

5.8.1 对文件的删除条件进行检查

实例3中*unlink*（）函数最终映射到*sys_unlink*（）系统调用函数去执行。先获取*hello.txt*文件的*i*节点，之后检查*i*节点属性信息、当前进程对该文件的操作权限等信息，确定该文件是否能够删除。

执行代码如下：

```
//代码路径： fs/namei.c:

int sys_unlink (const char * name)

{
```

```

const char * basename;

int namelen;

struct m_inode * dir, *inode;

struct buffer_head * bh;

struct dir_entry * de;

    if ( ! (dir=dir_namei (name, &namelen, &basename) ) ) //通
    过分析路径名，找到将要删除文件的枝梢i节点

        return-ENOENT;

    if ( ! namelen ) { //如果namelen为0

        iput (dir) ; //释放枝梢i节点

        return-ENOENT;

    }

    if ( ! permission (dir,MAY_WRITE) ) { //如果用户进程没有枝梢i
    节点对应目录文件的写入权限

        iput (dir) ; //释放枝梢i节点

        return-EPERM;

    }

    bh=find_entry (&dir,basename,namelen, &de) ; //获得目标文件
    的目录项

```

```

if (! bh) {

    iput (dir) ;

    return-ENOENT;

}

    if (! (inode=iget (dir->i_dev,de->inode) ) ) { //获得要删除文件的i节点

        iput (dir) ;

        brelease (bh) ;

        return-ENOENT;

    }

    if ( (dir->i_mode&S_ISVTX) && ! suser () &&

        current->euid !=inode->i_uid&&

        current->euid !=dir->i_uid) { //如果用户进程不具备删除该文件的权限

        iput (dir) ; //释放枝梢i节点

        iput (inode) ; //释放目标文件i节点

        brelease (bh) ;

        return-EPERM;

```

```

    }

    if (S_ISDIR (inode->i_mode) ) { //如果目标文件是个目录文件

        iput (inode) ; //释放目标文件i节点

        iput (dir) ; //释放枝梢i节点

        brelease (bh) ;

        return-EPERM;

    }

    if (! inode->i_nlinks) { //如果该i节点的链接数为0（没有任何进程与之存在关联），则强行置1

        printk ("Deleting nonexistent file (%04x: %d) , %d\n",

            inode->i_dev,inode->i_num,inode->i_nlinks) ;

        inode->i_nlinks=1;

    }

    //以下为具体的文件删除工作

    de->inode=0;

    bh->b_dirt=1;

    brelease (bh) ;

```



```
inode->i_nlinks--;
```

```
inode->i_dirt=1;
```

```
inode->i_ctime=CURRENT_TIME;
```

```
iput (inode) ;
```

```
iput (dir) ;
```

```
return 0;
```

```
}
```

5.8.2 进行具体的删除工作

删除hello.txt文件的情景如图5-14所示。

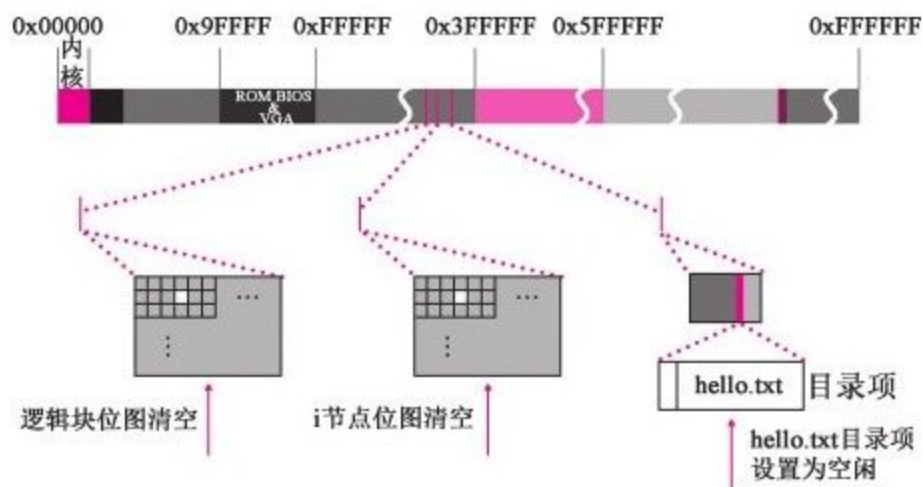


图 5-14 删除hello.txt文件

具体执行代码如下：

//代码路径：fs/namei.c:

```
int sys_unlink (const char * name)
```

```
{
```

```
const char * basename;
```

```
int namelen;
```

```
struct m_inode * dir, *inode;
```

```
struct buffer_head * bh;
```

```
struct dir_entry * de;
```

```
.....
```

```
//以下为具体的文件删除工作
```

```
de->inode=0; //将user2目录文件中hello.txt目录项清空
```

```
bh->b_dirt=1; //将hello.txt目录项所在缓冲块b_dirt置1
```

```
brelse (bh) ;
```

```
inode->i_nlinks--; //目标文件链接数递减，在实例3中，该值被递减为0
```

```
inode->i_dirt=1; //目标文件i节点i_dirt置为1
```

```
inode->i_ctime=CURRENT_TIME;
```

```
iput (inode) ; //释放hello.txt文件i节点
```

```
iput (dir) ; //释放user2目录文件i节点
```

```
return 0;
```

```
}
```

值得注意的是，`iput()` 函数中执行的情景与5.7.2节中的有所区别。

执行代码如下：

```
//代码路径：fs/open.c:
```

```
void iput (struct m_inode * inode) //释放目标文件i节点
```

```
{
```

```
.....
```

```
repeat:
```

```
if (inode->i_count > 1) { //i节点的引用计数大于1
```

```
inode->i_count--; //递减其引用计数
```

```
return;
```

```
}
```

```
if (! inode->i_nlinks) { //此时链接数已递减为0，说明没有进程  
与该i节点存在关联
```

```
truncate (inode) ; //根据i节点中i_zone[9], 释放文件在硬盘上占  
据的逻辑块
```

```
free_inode (inode) ; //将i节点位图中对应的位清空, 并将  
inode_table[32]中的表项清空
```

```
return;
```

```
}
```

```
if (inode->i_dirt) { //对于本案例, i节点内容已经改变, 同步该i  
节点内容到外设
```

```
write_inode (inode) ; /*we can sleep-so do again*/
```

```
wait_on_inode (inode) ;
```

```
goto repeat;
```

```
}
```

```
inode->i_count--; //i节点引用计数递减
```

```
return;
```

```
}
```

调用truncate () 函数, 根据文件i节点中
i_zone[9]释放文件在外设上的所有逻辑块。执行

代码如下：

```
//代码路径： fs/open.c:

void truncate (struct m_inode * inode)

{

    int i;

    if ( ! (S_ISREG (inode->i_mode) || S_ISDIR (inode->
i_mode) ) ) //如果hello.txt文件不是普通文件或目录文件

        return; //直接返回

    for (i=0; i<7; i++)

        if (inode->i_zone[i]) {

            free_block (inode->i_dev,inode->i_zone[i]) ; //将i_zone前7项逻辑块在逻辑块位图上对应的位清零

            inode->i_zone[i]=0;

        }

        free_ind (inode->i_dev,inode->i_zone[7]) ; //将一级间接块自身占用的逻辑块以及它管理的逻辑块在逻辑块位图上对应的位清零

        free_dind (inode->i_dev,inode->i_zone[8]) ; //将二级间接块自身占用的逻辑块以及它管理的逻辑块在逻辑块位图上对应的位清零
```

```
inode->i_zone[7]=inode->i_zone[8]=0;

inode->i_size=0;

inode->i_dirt=1;

inode->i_mtime=inode->i_ctime=CURRENT_TIME;

}
```

调用 `free_inode ()` 函数，清空 `i` 节点位图和 `i` 节点表项。

执行代码如下：

```
//代码路径： fs/bitmap.c:

void free_inode (struct m_inode * inode)

{

    struct super_block * sb;

    struct buffer_head * bh;

    if (! inode) //如果i节点为空
```

```

return;

if (! inode->i_dev) { //如果设备号为0

memset (inode, 0, sizeof (*inode) ) ;

return;

}

if (inode->i_count > 1) { //如果i节点被多次引用

    printk ("trying to free inode with count=%d\n", inode->
i_count) ;

    panic ("free_inode") ;

}

if (inode->i_nlinks) //如果i节点还与进程保持关系

    panic ("trying to free inode with links") ;

    if (! (sb=get_super (inode->i_dev) ) ) //如果i节点所在文件
系统的超级块不存在

        panic ("trying to free inode on nonexistent device") ;

    if (inode->i_num < 1 || inode->i_num > sb->s_ninodes) //检查i节
点号

        panic ("trying to free inode 0 or nonexistent inode") ;

```



```

    if ( ! (bh=sb->s_imap[inode->i_num >> 13]) ) //如果该i节点
对应的i节点位图不存在

    panic ("nonexistent imap in superblock") ;

    if (clear_bit (inode->i_num&8191, bh->b_data) ) //清空i节点
位图中与hello.txt文件i节点对应的位

    printk ("free_inode: bit already cleared.\n\r") ;

    bh->b_dirt=1; //将i节点位图所在缓冲块的b_dirt置1（表示需要同
步）

    memset (inode, 0, sizeof (*inode) ) ; //将i节点表中hello.txt文
件i节点的表项清零

}

```

操作系统将被清空的i节点位图、逻辑块位图、i节点表项信息，同步到硬盘上（并未清除对应的逻辑块中的内容）。它们都是hello.txt文件的管理信息，这些信息不存在了，即便该文件的逻辑块内容还存储在硬盘上，也无法再访问到该文件。

5.9 本章小结

本章通过几个实例程序，详细讲解了与文件操作相关的代码和知识。

操作系统对文件的一切操作，都可以分为两个方面：对super_block、d_super_block、m_inode、d_inode、i节点位图、逻辑块位图这类文件管理信息的操作以及对文件数据内容的操作。新建、打开、关闭、删除文件属于对文件管理信息的操作。读文件、写文件和修改文件则主要是操作文件数据内容。

操作文件管理信息就是建立或解除进程与文件的关系链条，链条的主干为task_struct中的*filp[20]——file_table[64]——inode_table[32]。进

程就可以沿着关系链条，依托缓冲区与硬盘进行数据交互。当关系链条解除后，进程则不再具备操作指定文件的能力。如果文件管理信息被更改，则操作系统要将此更改落实在硬盘上，以免失去对文件数据内容的控制。

第6章 用户进程与内存管理

现代操作系统的重要特征就是支持实时多任务——同时运行多个程序。运行中的程序被称为进程。在类UNIX操作系统的设计者看来，操作系统的核心就是进程。所谓的操作系统就是若干个正在运行、操作的进程构成的系统。按照这个思路，进程的创建只可能由进程承担，也就是父子进程创建机制。在任何情况下，至少得有一个进程留守，这就是进程0。与计算机使用者交互也是由专门的进程（即shell）负责。总之，一切皆为进程。

在一台计算机只有一个CPU、一个CPU只有一个核的时代，多个进程同时运行的本质是分时

轮流运行。确保多进程同时正确运行，就必须解决两个关键问题：一个是如何防止多进程同时运行时，一个进程的代码、数据不会被其他进程直接访问、覆盖；另一个是如何做到多进程有序轮流执行。

第一个问题涉及进程保护，第二个问题涉及进程调度。

6.1 线性地址的保护

在Intel IA-32架构中，进程保护体现在对进程内存空间的保护，进程内存空间的保护是由线性地址保护、物理地址保护实现的。

现在计算机基本上都沿用冯·诺依曼体系。这个体系中，指令、数据都存储在同样的内存中。

内存设计为随机访问存储器（RAM），也就是说可以在内存空间任意读、写数据或指令。在没有保护模式之前，从物理内存的角度看，不同用户程序的代码和数据没有物理上的差异，都由一连串的0、1组成。至于什么地方可以读写，什么地方不可以读写，并没有明确的物理限制，没有什么机制能阻拦不同用户程序间的相互干扰。

支持实时多任务首先遇到的问题就是，如何保证每个进程在运行过程中能够与其他进程互不干扰，也就是保证进程之间不能相互访问代码、数据，更不能相互覆盖代码、数据。

6.1.1 进程线性地址空间的格局

Intel IA-32 CPU架构设计为，只要打开PE、PG，所有在计算机中运行的程序使用的只能是线性地址，然后将线性地址转换到具体的物理地址，转换是由CPU中的MMU根据页目录表、页表、页的设定，由硬件自动实现的。

线性地址就是CPU可以寻址的地址。在IA-32体系架构下，32位地址总线的线性地址空间范围是0~4 GB。为了在线性地址层面分隔进程的内存空间，Linux 0.11采取的总体策略是将4 GB的线性地址空间分成互不重叠的64等份，每份64 MB，每个进程一份，最多同时开启64个进程。要求进程无论怎样执行，都不能跨越起点和终点。这样进程的线性地址空间彼此不重叠，实现在线性地址层面对进程内存空间的保护。这是整个

Linux 0.11中线性地址空间设计的最大格局，所有针对线性地址空间方面的设计都服从于这个格局。

`task[64]`是这个格局的基点，所有进程登记、注销都只由它统一管理。每个进程只有在`task[64]`中定位后，才能在线性地址空间中进行安排。操作系统根据`task[64]`的项号`nr`在GDT中找到对应的LDT。`task[64]`起到了控制进程总量，关联进程与GDT中的LDT、TSS的关键作用。

虽然在操作系统代码中规划出了64等分4 GB线性地址空间的格局，但仅此能否对进程跨越64 MB线性地址空间的访问做出有效的阻拦？也就是说，能否确保进程的线性地址空间彼此不重叠？

操作系统内核虽然做出了进程的线性地址空间的格局，但却无法仅仅依靠算法、控制逻辑阻拦进程的跨界访问。原因是CPU一次只能执行一条指令，执行进程的指令就不能执行内核的指令。所以，不论内核有多么漂亮的控制越界算法，当进程执行跨界访问时，内核的控制越界算法都不在执行状态，无法控制进程的越界行为。

软件的方法无效，那只能依靠硬件方法。

Intel IA-32架构专门设计了基于CPU硬件的控制进程访问越界的方法。

6.1.2 段基址、段限长、GDT、LDT、特权级

Intel IA-32架构对进程线性地址空间的保护是基于段的。

历史上，由于函数（子程序）链接的需要，发明了在内存中划分段的方法，所有程序的设计都是基于段的。线性地址空间是一维的，所以只要看住一段线性地址空间的两头，让程序在段空间里执行，别越界，它就不会干扰到其他段，也就不会干扰到其他进程。

Intel早期的CPU为了降低成本，只设计了看住段起始位置的段头寄存器，并没有设计看住段

结束位置的段尾寄存器。为了兼容早期CPU,Intel IA-32架构在段（头）寄存器中设计了段限长，等效于设计了段尾寄存器，用一个寄存器巧妙地起到了两个寄存器的作用。

Linux 0.11操作系统利用Intel IA-32 CPU架构提供的段基址、段限长有效地阻拦了段内跳转中有意无意的越界行为。比如，`jmp X`，如果这个X很大，超过段限长，硬件会阻止类似指令的执行，并立即报出GP错误。

对于进程代码中跨越段边界的`ljmp`，段基址、段限长不能阻拦，Linux 0.11是用什么方法拦截非法跨越进程边界的访问动作的呢？

非法跨越进程边界有两种情况：一种是从一个进程非法跨越到另一个进程，另一种是从一个进程非法跨越到内核。

1. 从一个进程非法跨越到另一个进程

从一个进程用ljmp指令非法跨越到另一个进程，从IA-32架构的角度看，两个进程的代码段都是3特权级，Linux 0.11的所有进程都安排在一个4 GB的线性地址空间，所以允许这个ljmp指令的执行，段基址、段限长此时起不到阻拦非法越界的作用。Linux 0.11采用的是通过LDT的设计，阻拦非法ljmp指令的执行。

第2章中讲解了Linux 0.11的GDT、LDT的设计。64个进程，每个进程占用GDT的两项，一项

是TSS，另一项就是LDT。所有进程的LDT段的设计是完全一样的，每个LDT都有3项，都是第一项为空，第二项是进程代码段，第三项是进程数据段。当一个进程的代码中有非法的跨进程跳转的指令时，比如，`ljmp`指令执行时，该指令后面的操作数是“段内偏移段选择子”。代码段的段选择子存储在CS里面。仔细考察一下，可以看出Linux 0.11中所有进程的CS的内容都是一样的，用二进制表示的形式都是00000000000001111。CPU硬件无法识别是哪一个进程的CS，也就无法选择段描述符，只能默认使用当前LDT中提供的段描述符，所以类似`ljmp`这样的段间跳转指令，无论后面操作数怎么写，都无法跨越当前进程的代码段，也就无法进行段间跳转，最终只能是执

行到本段。由此可见，Linux 0.11的LDT的设计看似重复，其实颇具匠心。

试想一下，如果Linux 0.11不是这样的设计，而是将所有进程的代码段描述符都直接写到GDT中。对所有进程共用一个4 GB线性地址空间的Linux 0.11而言，进程代码中的非法跨越进程的跳转指令就可以不受阻拦地执行。

按照这个思路，结合第2章讲解的内容仔细思考，可以发现，Linux 0.11在防止非法跨越进程的长跳转指令方面，略显粗糙。第2章中讲解过TSS段、LDT段的段限长是一样的，都是104 B。这个段限长对TSS来说是合适的，对LDT来说就太长了。LDT只有3项，每项8字节，一共只有24 B。从进程0的INIT_TASK可以看出LDT后面紧跟着

TSS，这个数据结构在父子进程创建机制创建进程时会向后遗传，所有进程的task_struct里面的LDT、TSS都是一样的。如果进程代码中有这样的代码：

ljmp偏移，CS（CS的值是0000000000111111，即3特权级，LDT表中的第8项）

这样的指令仍然会在段内执行，而从LDT基址往后偏移到“第8项”的数据内容不可预知，出现的错误也不可预知。但无论是哪些错误，都无法跨越进程的边界，也无法改变LDT。

反过来看这个问题，就算非法的进程跨越的长跳转指令能够执行，也只是代码跳转过去，数据段、栈段都没跟着变换过去。代码在一个进程

的段中执行，数据和栈却在另一个进程中，在这种条件下，代码通常会执行死了。从这个反向角度，我们可以更深刻地领悟到为什么正常的进程切换，是用TSS将代码段、数据段、栈全部变换过去的。要想进行正确的进程切换，必须将进程的执行状态成套、完整地保存，并成套、完整地切换到另一个进程。

以上讲解的是用ljmp从一个进程非法跨越到另一个进程的情况，下面讨论用ljmp从一个进程非法跨越到内核的情况。

2. 从一个进程非法跨越到内核

用户进程代码段的特权级都是3，内核的特权级是0，Intel IA-32架构禁止代码跨越特权级长跳

转，3特权级长跳转到0特权级是禁止的，0特权级长跳转到3特权级同样是禁止的。所以这样的非法长跳转指令会被CPU硬件有效阻拦，进程与内核的边界得到有效的保护。0特权级的内核代码可以访问3特权级的进程数据，3特权级的进程代码不能访问0特权级的进程代码。这些禁止都是非常刚性的硬件禁止。

从上面的讲解可以看出，Linux对GDT、LDT的设置，有效地阻止了非法跨越进程边界的访问。用户进程是否可以设置GDT、LDT，以使自己写的非法跨越边界的指令能够执行？答案是否定的，因为Linux 0.11将GDT、LDT这两个数据结构设置在内核数据区，是0特权级的，只有0特权级的代码才能修改设置GDT、LDT。

那么，用户进程是否可以在自己的数据段按照自己的意愿重新做一套GDT、LDT呢？如果仅仅是形式上做一套和GDT、LDT一样的数据结构，没有什么不可以，但起不到真正的GDT、LDT的作用。真正起作用的GDT、LDT是CPU硬件认定的。这两个数据结构的首地址必须挂接在CPU中的GDTR、LDTR上，运行时，CPU只认GDTR、LDTR指向的数据结构，其他数据结构就算起名字叫GDT、LDT,CPU也一概不认。Linux 0.11内核在进程的初始化阶段，就将GDT、LDT挂接到了CPU中的GDTR、LDTR上了。

用户进程能否也将自己制作的GDT、LDT挂接到GDTR、LDTR上？答案是否定的，因为对

GDTR、LDTR进行设置的指令LGDT、LLDT只能在0特权级下执行。

到此为止，我们可以看清楚Linux操作系统依托Intel IA-32架构设计的段基址、段限长、GDT、LDT、特权级这一整套硬件保护机制，在线性地址层面建立了牢固的进程间、进程与内核间的边界，有效地防止了非法跨越边界的操作。

进程间合理的跨越边界的数据沟通如何解决？进程间切换及进程需要跨越边界获得操作系统内核合理的支持应该如何操作呢？

第一个问题，将在第8章中讲解。

第二个问题，涉及TSS及CPU硬件的中断门。

Linux 0.11中的进程间切换是在`schedule()`中完成的，其技术路线很像任务门（但没有用任务门），是在0特权级下，用`ljmp`指令直接跳转到要切换的进程的TSS（指令跳转到数据，似乎有些奇怪，实际上后面有CPU硬件做了大量的工作，最终跳转到目标进程的代码段），实现进程切换的。

进程要想获得内核的支持（如读盘），Intel IA-32架构提供了中断门技术，支持由3特权级代码段翻转到0特权级代码段执行。注意，这个翻转不是普通的跳转，需要经过CPU的硬件中断机制，不同于平坦的、普通的内存寻址跳转。获得内核的支持后，再由`iret`指令经过CPU硬件，从0

特权级的内核代码段翻转到3特权级的进程代码段继续执行。

6.2 分页

6.2.1 线性地址映射到物理地址

前面我们介绍了线性地址，线性地址最终要转换为物理地址。Linux 0.11在启动前打开了PG，线性地址是通过页目录表——页表——页面三级映射模式，最终落实到物理地址的。代码如下：

```
//代码路径: boot/head.s:

.....

xorl %eax, %eax/*pg_dir is at 0x0000*/

movl %eax, %cr3/*cr3-page directory start*/

movl %cr0, %eax

orl $0x80000000, %eax
```

```
movl %eax, %cr0/*set paging (PG) bit*///设置CR0, 打开PG  
ret/*this also flushes prefetch-queue*/  
  
.....
```

通过第1章的介绍我们得知，在打开PG前，已经打开了PE，转入保护模式运行。CPU的硬件默认，在保护模式下，如果没有打开PG，线性地址恒等映射到物理地址；如果打开了PG，则线性地址需要通过MMU进行解析，以页目录表、页表、页面的三级映射模式映射到物理地址。

保护模式下，是否打开PG对线性地址映射到物理地址的影响如图6-1所示。

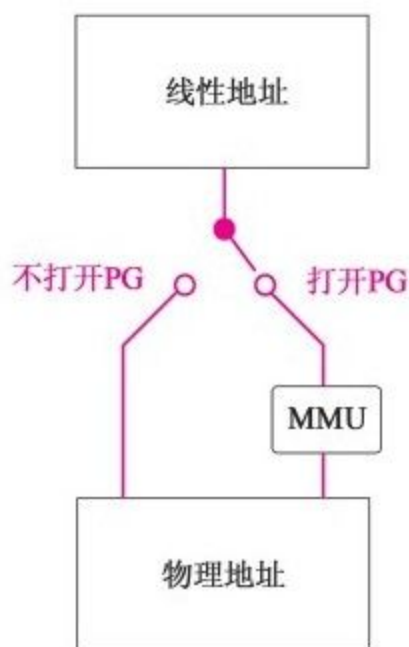


图 6-1 PG影响映射的情景

Linux 0.11为什么要打开PG呢？6.1节我们已经介绍过，IA-32体系下，线性地址空间范围是0～4 GB，已经被64个进程平分了，各自64 MB。如果不打开PG，根据CPU默认的规则，线性地址就只能直接映射到物理地址，而Linux 0.11最大只能支持16 MB的物理内存，显然绝大部分线性地址空间都作废了，无法支持多进程同时执行，所

以要打开PG，将进程的线性地址，根据物理内存的实际承载能力，有秩序地映射到物理地址上，以此支持多进程执行。

线性地址映射到物理地址的过程是这样的：
每个线性地址值是32位，MMU按照10—10—12的长度，来识别线性地址值，并分别将其解析为页目录项号、页表项号、页面内偏移，最终映射到物理地址。这个过程示意图如图6-2所示。

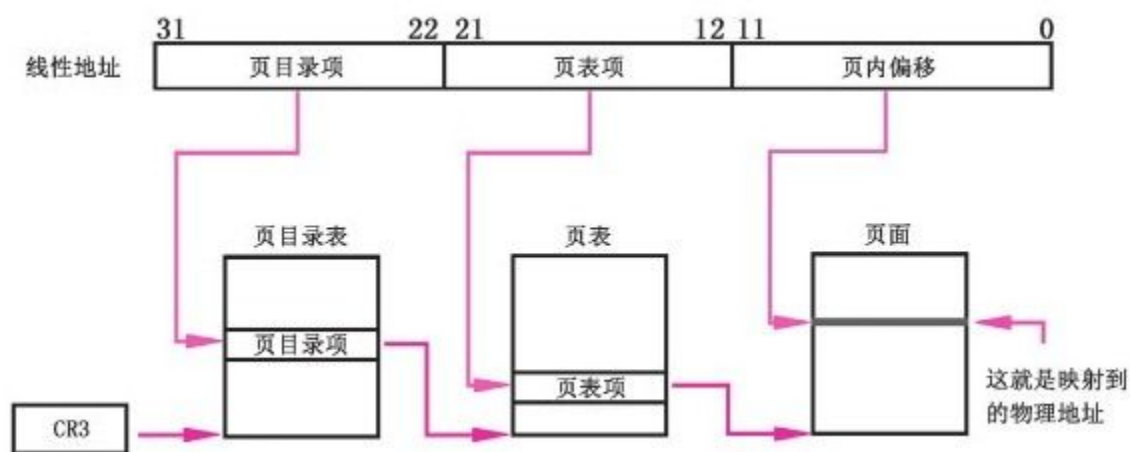


图 6-2 页目录表、页表、页面的映射情景

Linux 0.11中只有一个页目录表，CR3中存储着页目录表的基址，这样MMU解析线性地址时，先找CR3中的信息，这样就可以找到页目录表。只有找到了页目录表，才会有下面的解析，所以在打开PG前，最重要的事情是要把页目录表的基址载入CR3。代码如下：

```
//代码路径: boot/head.s:

.....

stosl/*fill pages backwards-more efficient: -) */

subl $0x1000, %eax

jge 1b

xorl %eax, %eax/*pg_dir is at 0x0000*/

movl %eax, %cr3/*cr3-page directory start*///将页目录表的基址0
载入CR3

movl %cr0, %eax

orl $0x80000000, %eax
```

```
movl %eax, %cr0/*set paging (PG) bit*///设置CR0, 打开PG
```

```
ret/*this also flushes prefetch-queue*/
```

```
.....
```

通过解析线性地址值中表示页目录项的10位数据，就可以在页目录表中找到页目录项。页目录项里面记录着页表的物理地址值，可以据此找到页表的位置，再通过解析线性地址值中表示页表项的10位数据找到页表项。同样，页表项中记录着表示页面的物理地址值，可以据此找到页面的位置，之后再分析线性地址值中表示页内偏移的12位物理地址值，最终就找到了物理地址。

页目录表、页表、页面的三级映射关系是由内核建立的。内核建立映射关系时，可以让不同的线性地址映射到不同的物理地址，也可以映射

到相同的物理地址。下面我们先以进程执行时分页为例，介绍不同的线性地址映射到不同的物理地址的情况。

6.2.2 进程执行时分页

分页以及映射物理页面时，内核需要做到以下几点。

1) 只能从空闲页面中分配新页面，不能分配其他进程正在使用的页面，干扰其他进程的执行，更不能将内核区域的页面挪作他用。

从第2章的介绍我们得知，急速前内核通过 `mem_map` 结构对 1 MB 以上的内存空间进行分页管理，而且主内存中每个页面的引用计数都被初始化为 0，即默认为空闲页面。代码如下：

```
//代码路径: mm/memory.c:
```

```
.....
```

```

#define LOW_MEM 0x100000//1 MB

#define PAGING_MEMORY (15*1024*1024)

#define PAGING_PAGES (PAGING_MEMORY >> 12) //页面数
量

#define MAP_NR (addr) ( ( (addr) -LOW_MEM) >> 12)

#define USED 100

.....

void mem_init (long start_mem,long end_mem)

{

int i;

HIGH_MEMORY=end_mem;

for (i=0; i<PAGING_PAGES; i++)

mem_map[i]=USED;

i=MAP_NR (start_mem) ;

end_mem-=start_mem;

end_mem>>=12;

while (end_mem-->0) //能够占用的页面，引用计数设置为0

```

```
mem_map[i++]=0;

}
```

为进程实际分配页面时，只在mem_map的控制范围内操作，而且只选择引用计数为0的页面。如果申请到，就将引用计数置1，防止挪作他用，引起混乱。代码如下：

```
//代码路径：mm/memory.c:
```

```
.....
```

```
#define PAGING_MEMORY (15*1024*1024) //1 MB以下的不进行分页管理
```

```
.....
```

```
#define MAP_NR (addr) ( ( (addr) -LOW_MEM) >> 12)
```

```
.....
```

```
unsigned long get_free_page (void)
```

```
{
```

```
register unsigned long __res asm ("ax") ;
```

```
__asm__ ("std; repne; scasb\n\t"//只选择引用计数为0的页面
```

```
"jne 1f\n\t"
```

```
"movb$1, 1 (%%edi) \n\t"//申请后，引用计数置为1
```

```
"sall$12, %%ecx\n\t"
```

```
"addl%2, %%ecx\n\t"
```

```
.....
```

```
"1: "
```

```
: "=a" (__res)
```

```
: "0" (0) , "i" (LOW_MEM) , "c" (PAGING_PAGES) ,
```

```
"D" (mem_map+PAGING_PAGES-1) //在mem_map[]的管理范围  
内查找空闲页
```

```
: "di", "cx", "dx") ;
```

```
return __res;
```

```
}
```

如果分配不到页面，要强行干预，终止程序继续执行。代码如下：

//代码路径: kernel/fork.c:

```
int copy_process (int nr,long ebp,long edi,long esi,long gs,long
none,

long ebx,long ecx,long edx,

long fs,long es,long ds,

long eip,long cs,long eflags,long esp,long ss)

{

.....

struct task_struct * p;

int i;

struct file * f;

p= (struct task_struct *) get_free_page () ; //为进程task_struct和
内核栈分配空闲页面

if (! p) //如果分配不到页面，就返回错误信息，终止程序执行

return-EAGAIN;
```

```
task[nr]=p;
```

```
*p=*current; /*NOTE ! this doesn't copy the supervisor stack*/
```

```
.....
```

```
}
```

```
int copy_page_tables (unsigned long from,unsigned long to,long  
size)
```

```
{
```

```
unsigned long * from_page_table;
```

```
unsigned long * to_page_table;
```

```
unsigned long this_page;
```

```
unsigned long * from_dir, *to_dir;
```

```
unsigned long nr;
```

```
{
```

```
.....
```

```
if ( ! (1&*from_dir) )
```

```
continue;
```

```
from_page_table= (unsigned long *) (0xfffff000&*from_dir) ;
```

```
    if ( ! (to_page_table= (unsigned long *) get_free_page ( ) ) ) //  
为复制页表申请空闲页面
```

```
    return -1; /*Out of memory,see freeing*///如果申请不到，返回-1，  
终止程序执行
```

```
    *to_dir= ( (unsigned long) to_page_table) |7;
```

```
    nr= (from==0) ?0xA0: 1024;
```

```
    .....
```

```
}
```

```
void do_no_page (unsigned long error_code,unsigned long address)
```

```
{
```

```
    .....
```

```
    if (share_page (tmp) )
```

```
    return;
```

```
    if ( ! (page=get_free_page ( ) ) ) //为进程加载程序申请空闲页  
面
```

```
    oom ( ) ; //如果申请不到，就强行让进程退出
```

```
    /*remember that 1 block is used for header*/
```

```
    block=1+tmp/BLOCK_SIZE;
```

```
    for (i=0; i<4; block++, i++)
```

```
nr[i]=bmap (current->executable,block) ;
```

```
.....
```

```
}
```

```
static inline volatile void oom (void)
```

```
{
```

```
printk ("out of memory\n\r") ;
```

```
do_exit (SIGSEGV) ; //进程退出
```

```
}
```

```
#define PAGING_MEMORY (15*1024*1024) //1 MB以下的不进行分页管理
```

```
.....
```

```
#define MAP_NR (addr) ( ( (addr) -LOW_MEM) >> 12)
```

```
.....
```

```
unsigned long get_free_page (void)
```

```
{
```

```
register unsigned long __res asm ("ax") ;
```

```
__asm__ ("std; repne; scasb\n\t"
```

```

"jne 1f\n\t"

.....

"1: "

: "=a" (__res)

: "0" (0) , "i" (LOW_MEM) , "c" (PAGING_PAGES) ,

"D" (mem_map+PAGING_PAGES-1) //在mem_map[]的管理范围
内查找空闲页

: "di", "cx", "dx" ) ;

return __res;

}

```

2) 要明确什么时候该为进程新申请页面，什么时候不该申请。

通过第1章的介绍我们得知，每个页目录项和页表项的最后3位，标志着其所管理的页面的属性（一个页表本身也占用一个页面），它们分别是

U/S、R/W和P。判断是否该申请页面，是在解析线性地址时确定的，关键要看P这个标志位。

一个页目录项或一个页表项，如果和一个页面建立了映射关系，P标志就设置为1；如果没建立映射关系，该标志就是0。进程执行时，线性地址值都会被MMU解析。如果解析出某个表项的P位为0，说明该表项没有对应页面，就会产生缺页中断。前面我们所说的缺页中断，就是这样产生的。如果P位为1，就说明该表项对应着具体的页面，直接根据表项中记录的地址值找到具体的页面。所以，这一位非常重要，设计者在设计内核时，始终都要保证这一位的信息明确，绝对不能出现垃圾值。因为垃圾值就等于错误。

内核分页时，就是先把一个页目录表和4个页表全部清零，然后把P位设置为1。代码如下：

```
//代码路径: boot/head.s:
```

```
.....
```

```
setup _paging: //严格的清零操作
```

```
movl $1024*5, %ecx/*5 pages-pg_dir+4 page tables*/
```

```
xorl %eax, %eax
```

```
xorl %edi, %edi/*pg_dir is at 0x000*/
```

```
cld; rep; stosl
```

```
movl $pg0+7, _pg_dir/*set present bit/user r/w*/
```

```
movl $pg1+7, _pg_dir+4/*-----"-----*/
```

```
movl $pg2+7, _pg_dir+8/*-----"-----*/
```

```
movl $pg3+7, _pg_dir+12/*-----"-----*/
```

```
movl $pg3+4092, %edi
```

```
movl $0xfff007, %eax/*16Mb-4096+7 (r/w user,p) */
```

```
std
```

```
1: stosl/*fill pages backwards-more efficient: -) */  
  
subl $0x1000, %eax  
  
jge 1b  
  
.....
```

7的二进制形式是111，P被设置为1了。

创建进程的时候，会申请页面。只要调用 `get_free_page()` 函数，就要把内存清零，因为无法预知这页内存的用途。如果是用作页表，不清零就有垃圾值，就是隐患。代码如下：

```
//代码路径: mm/memory.c:  
  
unsigned long get_free_page (void)  
  
{  
  
.....  
  
"leal 4092 (%%edx) , %%edi\n\t"
```



```
"rep; stosl\n\t"//页面清零
```

```
.....
```

```
}
```

复制页表时，就得建立映射关系。关系建立后，就把P位设置为1。代码如下：

```
//代码路径： mm/memory.c:
```

```
int copy_page_tables (unsigned long from,unsigned long to,long  
size)
```

```
{
```

```
.....
```

```
from_page_table= (unsigned long *) (0xfffff000&*from_dir) ;
```

```
if (! (to_page_table= (unsigned long *) get_free_page () ) )
```

```
return-1; /*Out of memory,see freeing*/
```

```
*to_dir= ( (unsigned long) to_page_table) |7; //页目录项的P位被  
设置为1
```

```
nr= (from==0) ?0xA0: 1024;
```

```
for (; nr-- > 0; from_page_table++, to_page_table++) {  
  
    this_page = *from_page_table;  
  
    if (! (1 & this_page) )  
  
        continue;  
  
    this_page &= ~2; //页表项中的P位被设置为1, ~2的二进制形式  
    为101  
  
    *to_page_table = this_page;  
  
    if (this_page > LOW_MEM) {  
  
        *from_page_table = this_page;  
  
        .....  
    }  
}
```

页表和页面的关系解除后，页表项就要清零。页目录项和页表解除关系后，页目录项也要清零，这样就等于把对应的页表项、页目录项的P清零了。代码如下：

//代码路径: mm/memory.c:

```
int free_page_tables (unsigned long from,unsigned long size)

{

.....

if (1&*pg_table)

free_page (0xfffff000&*pg_table) ;

*pg_table=0; //页表项清零

pg_table++;

}

free_page (0xfffff000&*dir) ;

*dir=0; //页目录项清零

.....

}}
```

比如在进程加载程序阶段, 就调用了
free_page_tables () 函数把对应的页表项、页目

录项的P清零了，这样，当前进程线性地址对应的页面不存在，进程开始执行程序时，必产生缺页中断。

进程加载程序后，与新页面建立了映射关系，P位被设置成1，代码如下：

//代码路径：mm/memory.c:

```
unsigned long put_page (unsigned long page,unsigned long address)
{
    .....

    if ( (*page_table) &1)

    page_table= (unsigned long *) (0xfffff000&*page_table) ;

    else{

        if (! (tmp=get_free_page () ) )

        return 0;

        *page_table=tmp|7; //页目录项的P位被设置为1
```

```
page_table= (unsigned long *) tmp;

}

page_table[ (address >> 12) & 0x3ff]=page|7; //页表项中的P位
被设置为1

.....

}
```

3) 为进程新申请的页面要映射到该进程的线性地址空间内。

Linux 0.11把分页的基础建立在分段的基础上。每个进程的线性地址空间只要被限定住，彼此不干扰，那么分页时就不会出现混乱。前面已经介绍过进程线性地址空间的总体格局，将**IA-32**体系下的**4 GB**线性地址空间分为**64**等份，每个进程一份，彼此不干扰，页目录表的设计完全遵照这个格局。**Linux 0.11**的页目录表只有一个，一个

页目录表可以掌控1024个页表，一个页表掌控1024个页面，一个页面4 KB，这样一个页目录表就可以掌控 $1024 \times 1024 \times 4 \text{ KB} = 4 \text{ GB}$ 大小的内存空间。把一个页目录表64等分，每个进程可以占用16个页目录项，掌控16个页表，即占用64 MB的物理页面。这使得为每个进程分的页面，都可以映射到不同的页表项上，页表也映射到不同的页目录项上，MMU解析任何进程的线性地址时，最终都可以映射到不同的物理地址。当然，出于共享页面的需要，不同的线性地址是允许映射到相同的页面的，但那只不过是实际应用的需求，是一种策略。页目录表、页表、页面这套映射模式足以支持唯一的页面映射到唯一进程的线性地址空间。

下面我们介绍共享页面的问题。

6.2.3 进程共享页面

为进程分页时，每个进程的页面都会映射到进程自己的线性地址空间内，这样进程执行起来彼此不会干扰。但在有些情况下，进程需要共享页面，比如父子进程需要共享页面。最明显的例子就是进程1创建进程2后，进程2加载shell前，都和进程1共用代码，如下所示：

```
//代码路径：init/main.c:

void init (void)

{

.....

if ( ! (pid=fork ( ) ) ) {

close (0) ; //这些代码都是和进程共用的

if (open ("/etc/rc", O_RDONLY, 0) )
```



```
_exit (1) ;

execve ("/bin/sh", argv_rc,envp_rc) ;

_exit (2) ;

}

if (pid>0)

while (pid! =wait (&i) )

.....

}
```

此时最好的选择就是，子进程创建完毕后，先沿用着父进程的代码，父进程有多少页面，子进程就共享多少，将来子进程加载了自己的程序，再重新映射。这就引出了一个问题：多进程操作同一个页面，有读有写，这相当于给进程的封闭环境开了口子。为此，需要另想办法把这个口子堵住。

Linux 0.11用页表中的U/S、R/W两个位将这个口子堵住了。

先介绍U/S位。如果U/S位设置为0，表示段特权级为3的程序不可以访问该页面，其他特权级都可以；如果被设置为1，表示包括段特权级为3在内的所有程序都可以访问该页面。它的作用就是看死用户进程，阻止内核才能访问的页面被用户进程使用。当然，Linux 0.11中的保护，更偏重于使用“段”。

在怠速前为内核分页的时候，U/S位被设置为1，代码如下：

```
//代码路径: boot/head.s:
```

```
.....
```

```
setup _paging:
```

movl \$1024*5, %ecx/*5 pages-pg_dir+4 page tables*/

xorl %eax, %eax

xorl %edi, %edi/*pg_dir is at 0x000*/

cld; rep; stosl

movl \$pg0+7, _pg_dir/*set present bit/user r/w*/

movl \$pg1+7, _pg_dir+4/*-----"-----*/

movl \$pg2+7, _pg_dir+8/*-----"-----*/

movl \$pg3+7, _pg_dir+12/*-----"-----*/

movl \$pg3+4092, %edi

movl \$0xfff007, %eax/*16Mb-4096+7 (r/w user,p) */

std

1: stosl/*fill pages backwards-more efficient: -) */

subl \$0x1000, %eax

jge 1b

.....

代码中7的二进制形式是111，说明U/S位被设置为1。

创建进程的时候，子进程页目录项和页表项中的U/S位都被设置为1，代码如下：

//代码路径：mm/memory.c:

```
int copy_page_tables (unsigned long from,unsigned long to,long
size)
{
.....

from_page_table= (unsigned long *) (0xfffff000&*from_dir) ;

if (! (to_page_table= (unsigned long *) get_free_page () ) )

return-1; /*Out of memory,see freeing*/

*to_dir= ( (unsigned long) to_page_table) |7; //页目录项的U/S位
被设置为1

nr= (from==0) ?0xA0: 1024;

for (; nr-->0; from_page_table++, to_page_table++) {
```

```
this_page=*from_page_table;

if ( ! (1&this_page) )

continue;

this_page&=~2; //页表项中的U/S位被设置为1, ~2的二进制形式为101

*to_page_table=this_page;

if (this_page>LOW_MEM) {

*from_page_table=this_page;

.....

}
```

进程执行时，为进程新申请页面，并把页面映射到进程的线性地址空间，这会将页面对应的页表项的U/S位设置为1。如果是新申请的页表，也会将对应的页目录项U/S位设置为1。代码如下：

//代码路径: mm/memory.c:

```
unsigned long put_page (unsigned long page,unsigned long address)
```

```
{
```

```
.....
```

```
if ( (*page_table) &1)
```

```
page_table= (unsigned long *) (0xfffff000&*page_table) ;
```

```
else{
```

```
if (! (tmp=get_free_page () ) )
```

```
return 0;
```

```
*page_table=tmp|7; //页目录项的U/S位被设置为1
```

```
page_table= (unsigned long *) tmp;
```

```
}
```

```
page_table[ (address>>12) &0x3ff]=page|7; //页表项中的U/S  
位被设置为1
```

```
.....
```

```
}
```

下面介绍R/W位。如果它被设置为0，说明页面只能读不能写；如果设置为1，说明可读可写。

进程是可以共享页面的，这样会带来问题；如果多个进程往一个页面里面写数据，那么这个页面就会出现混乱。所以需要保护，R/W位就提供了这种保护。

创建进程的时候，父子进程共享页面。这些共享的页面就不能写入数据了，R/W位设置为0，代码如下：

```
//代码路径: mm/memory.c:

int copy_page_tables (unsigned long from,unsigned long to,long
size)
{
.....
```

```
this_page=*from_page_table;

if ( ! (1&this_page) )

continue;

this_page&=~2; //页表项中的R/W位被设置为0, ~2的二进制
形式为101

*to_page_table=this_page;

if (this_page>LOW_MEM) {

*from_page_table=this_page;

.....

}
```

再有，没有父子关系的两个进程也可以共享页面，不用另行加载，这时候也要把R/W位设置为0，禁止写入数据。代码如下：

//代码路径：mm/memory.c:

```
void do_no_page (unsigned long error_code,unsigned long address)
```



```

{

.....

if ( ! current->executable||tmp>=current->end_data) {

get_empty_page (address) ;

return;

}

if (share_page (tmp) )

return;

if ( ! (page=get_free_page ( ) ) )

oom ( ) ;

.....

}

static int share_page (unsigned long address) //准备共享页面

{

.....

if ( (*p) ->executable !=current->executable)

continue;

```

```
if (try_to_share (address, *p) )
```

```
return 1;
```

```
}
```

```
return 0;
```

```
}
```

```
static int try_to_share (unsigned long address, struct task_struct *  
p) //检测是否可以共享页面
```

```
{
```

```
.....
```

```
to&=0xfffff000;
```

```
to_page=to+ ( (address >> 10) & 0xffc) ;
```

```
if (1&* (unsigned long *) to_page)
```

```
panic ("try_to_share: to_page already exists") ;
```

```
/*share them: write-protect*/
```

```
* (unsigned long *) from_page&=~2; //页表项中的R/W位被设置  
为0, ~2的二进制形式为101
```

```
* (unsigned long *) to_page=* (unsigned long *) from_page;
```

```
invalidate () ;
```

.....

}

通过上述介绍不难发现，进程共享的页面，只可能有两种操作，要么读，要么写。读不会引起数据混乱，可以随便读。如果是写，就可能引起混乱，需要禁止。而对于写入的需求，Linux 0.11采取了一套写时复制的策略来解决，即把要写入数据的页面再复制一份给进程，两个进程各有一份，各写各的页面，这样就不会产生混乱。我们将在本章最后一节，用一个操作实例来讲解写时复制机制。

Linux 0.11中确有像管道这样的需求，两个进程在同一页面内又读又写。我们将在第8章中详细介绍如何保证进程操作管道时不产生数据混乱。

6.2.4 内核分页

进入保护模式后，内核先给自己分页。分页是建立在线性地址空间基础上的。前面一节我们介绍过，内核的段基址是0，代码段和数据段的段限长都是16 MB。每个页面大小为4 KB，每个页表可以管理1024个页面，每个页目录表可以管理1024个页表。既然确定了段限长是16 MB，这样就需要4个页目录项下辖4个页表，来管理这16 MB的内存，设置的代码如下：

```
//代码路径: boot/head.s:
```

```
.....
```

```
setup _paging:
```

```
movl $1024*5, %ecx/*5 pages-pg_dir+4 page tables*/
```

```

xorl %eax, %eax

xorl %edi, %edi/*pg_dir is at 0x000*/

cld; rep; stosl

movl $pg0+7, _pg_dir/*set present bit/user r/w*/

movl $pg1+7, _pg_dir+4/*-----"-----*/

movl $pg2+7, _pg_dir+8/*-----"-----*/

movl $pg3+7, _pg_dir+12/*-----"-----*/

movl $pg3+4092, %edi

movl $0xfff007, %eax/*16Mb-4096+7 (r/w user,p) */

std

1: stosl/*fill pages backwards-more efficient: -) */

subl $0x1000, %eax

jge 1b

.....

```

设置后对内存的管控情景如图6-3所示。

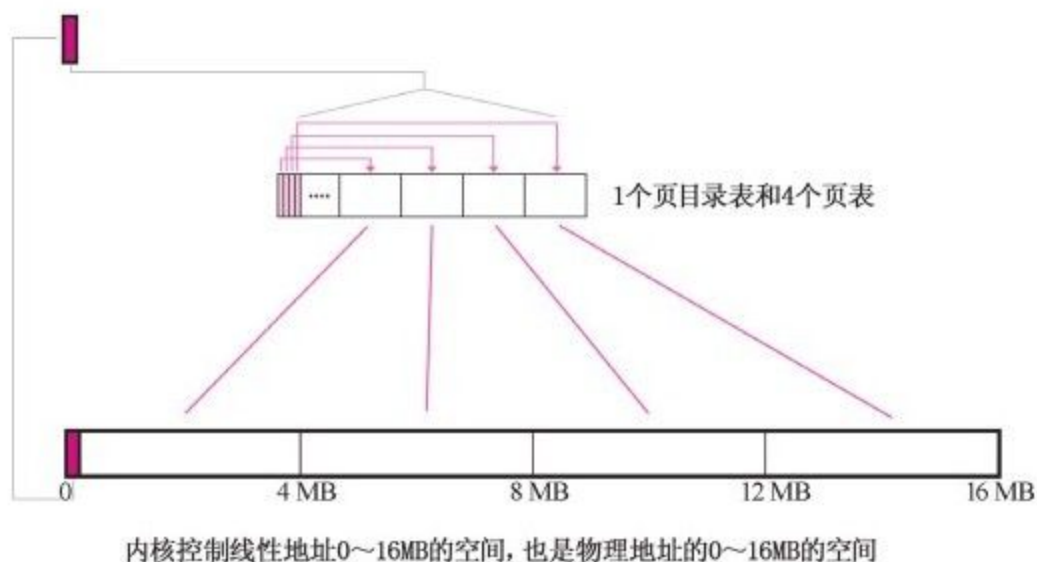


图 6-3 内存分页的情景

从图6-3中不难发现，内核的线性地址等于物理地址。这样做的目的是，内核可以对内存中的所有进程的内存区域任意访问。

可见恒等映射模式并不是唯一的模式，内核选择线性地址到物理地址的恒等映射，是因为对内核来讲最方便。比如，内核为进程申请了页面，这个页面总是要映射到页表项中，这需要往

页表项中写入该页面的物理地址。如果是恒等映射模式，调用`get_free_page()`函数后，获取的线性地址值直接就可以当物理地址来用，所以更为方便。

内核不仅掌控了所有内存页面的访问权，而且有权设置每个页面的读写、使用等属性，并把这些信息全部记录在页目录表和页表的表项中，代码如下：

```
//代码路径: boot/head.s:
```

```
.....
```

```
setup_paging:
```

```
movl $1024*5, %ecx/*5 pages-pg_dir+4 page tables*/
```

```
xorl %eax, %eax
```

```
xorl %edi, %edi/*pg_dir is at 0x000*/
```

```

cld; rep; stosl

movl $pg0+7, _pg_dir/*set present bit/user r/w*/

movl $pg1+7, _pg_dir+4/*-----"-----*/

movl $pg2+7, _pg_dir+8/*-----"-----*/

movl $pg3+7, _pg_dir+12/*-----"-----*/

movl $pg3+4092, %edi

movl $0xfff007, %eax/*16Mb-4096+7 (r/w user,p) *///可读写标志
置1, 保证内核可对该页面进行写入操作

std

1: stosl/*fill pages backwards-more efficient: -) */

subl $0x1000, %eax

jge 1b

.....

```

这些“7”的意义，我们在第1章中已经介绍了。

值得注意的是，CPU中硬件在解析线性地址值的时候，首先要能够找得到页目录表。如果找不到它，后续关于页表和页面的解析就无法进行。这个页目录表基址值，硬件默认保存在CR3里面。只要一解析线性地址值，就去CR3里面找基址，所以内核要把基址值载入CR3。代码如下：

```
//代码路径: boot/head.s:  
  
.....  
  
xorl %eax, %eax/*pg_dir is at 0x0000*/  
  
movl %eax, %cr3/*cr3-page directory start*/  
  
.....
```

这行对CR3操作的指令，只有0特权级的代码才能执行。这意味着，将来进程开始执行后，自

己另起一摊，模仿内核制作一套页目录表等数据结构。但由于3特权级下无法把这套结构挂接在CR3上，无法找到其他进程使用的内存页面，也就保护了其他进程。

另外值得注意的是，虽然内核的线性地址空间 and 用户进程不一样，内核是不能通过跨越线性地址空间直接访问进程的，但由于早就占有了所有的页面，而且特权级是0，所以内核执行时，可以对所有页面的内容进行改动，“等价于”可以操作所有进程所在的页面。但这与内核直接通过线性地址“段”来访问进程是两码事儿。一个典型的例子就是，某个进程要读盘，最终总要把缓冲区中的数据写到用户空间内，这件事要由内核来完成。代码如下：

//代码路径: mm/memory.c:

```
int file_read (struct m_inode * inode,struct file * filp,char * buf,int  
count)
```

```
{
```

```
.....
```

```
chars=MIN (BLOCK_SIZE-nr,left) ;
```

```
filp->f_pos+=chars;
```

```
left-=chars;
```

```
if (bh) {
```

```
char * p=nr+bh->b_data;
```

```
while (chars-->0)
```

```
put_fs_byte (* (p++) , buf++) ; //数据复制
```

```
br else (bh) ;
```

```
}else{
```

```
while (chars-->0)
```

```
put_fs_byte (0, buf++) ;
```

```
}
```

```
}
```

```
.....
```

```
}
```

```
//代码路径: include/asm/Segment.h:
```

```
extern inline void put_fs_byte (char val,char * addr)
```

```
{
```

```
    __asm__ ("movb%0, %%fs:  
%1": "r" (val) , "m" (*addr) ) ; //将一个字节存储在FS寄存器记  
录的段的内存地址中
```

```
}
```

```
//代码路径: kernel/system_call.s
```

```
.....
```

```
_system_call:
```

```
.....
```

```
movl $0x10, %edx#set up ds,es to kernel space
```

```
mov %dx, %ds
```

```
mov %dx, %es
```

```
movl $0x17, %edx#fs points to local data space
```

`mov %dx, %fs`//内核用FS寄存器存储用户进程LDT中的数据段描述符

`call _sys_call_table (, %eax, 4)`

.....

从以上代码可以看出，内核肯定是能直接访问进程的LDT对应的内存地址所在的页面，但不等于内核跨越了线性地址的段，访问了进程的线性地址空间，段的基础保护并没有因为分页而被打破。

6.3 一个用户进程从创建到退出的完整过程

根据前面讲解的原理，我们通过一个实例，理论联系实际，详细讲解一个用户进程从创建到退出的完整过程。

6.3.1 创建str1进程

1.为创建进程str1准备条件

首先我们来介绍一下str1进程的源代码，如下：

```
#include <stdio.h>
```

```
int foo (int n)
```

```
{

char text[2048];

if (n==0)

return 0;

else{

int i=0;

for (i; i< 2048; i++)

text[i]='\0';

printf ("text_%d=0x%x,Pid=%d\n", n,text,getpid () ) ;

sleep (5) ;

foo (n-1) ;

}

}

int main (int argc,char ** argv)

{

foo (6) ;

return 0;
```

```
}
```

硬盘上现在有一个叫做str1的可执行文件。用户在shell界面上输入一条指令

```
./str1, shell
```

程序会响应并解析这条指令，创建用户进程由此开始。

经解析得知，现在要执行str1这个进程，于是shell调用fork函数开始创建进程，产生int 0x80软中断，最终映射到sys_fork（）这个函数中，调用find_empty_process（）函数，为str1进程申请一个可用的进程号、在task[64]中为该进程申请一个空闲位置。我们这里假设str1这个进程是操作系统

怠速以后第一个申请的用户进程。通过前面章节的介绍可知，申请到的进程号是5，在task[64]中找到的空闲位置是第5项。

获取进程号和获取task[64]空闲项的情况，以及str1进程在task[64]中所占的位置如图6-4所示。

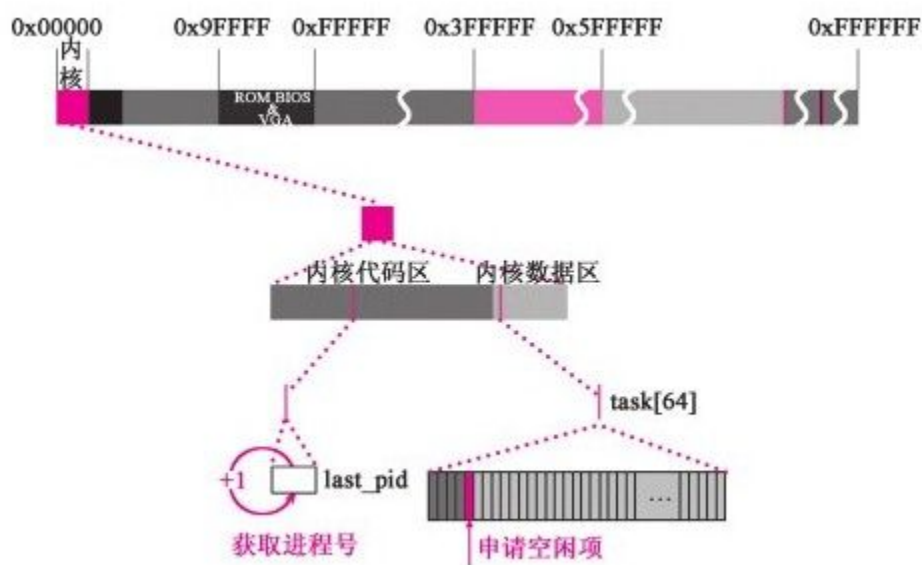


图 6-4 获取str1进程占用的进程号和task[64]中的位置

后面将根据task[64]中的项号，确定str1进程处于哪一个64 MB线性地址空间内，它的LDT和TSS将与GDT的哪两项挂接。我们来看后面的执行情况。

2.为str1进程管理结构找到存储空间

copy_process () 函数的第一件事就是为str1进程申请一个页面，这个页面用来承载进程的task_struct和内核栈。通过前面的介绍我们得知，为了实现对进程的保护，系统为每个进程的管理专门设计了一个结构，这就是task_struct。每个进程都有这样一本账，以此保证互不干扰。进程转入内核后，执行用的代码都是内核代码，但执行路径未必相同，导致数据压栈的顺序和内容不同。这些栈又不能存储在每个进程的用户空间

内，这样很容易被覆盖或改动，这就需要为每个进程专门准备一套内核栈。

通过前面讲解的内核分页策略可知，所有的页面，在刚进入保护模式时，就已经映射到了内核的16 MB线性地址空间。现在调用`get_free_page()`函数，是在内核中执行的，获得的用于`task_struct`和内核栈的页面只可能在内核的线性地址空间。从操作系统的后续程序中，也没有找到将这个页面映射到其他进程线性地址空间的代码。所以，尽管这个页面是为管理`str1`进程而分配的，但这个页面并没有映射到`str1`进程的线性地址空间，因此`str1`进程无法访问这个页面，这个页面始终掌握在内核手中。

从`get_free_page()` 函数申请页面的策略来看，操作系统要让进程使用的页面向高地址方向密排，以此提高内存的使用效率。进程执行时，尤其是多进程执行时，页面的释放是随机的。这就导致内存中经常会散布着被释放的空闲页面，而`get_free_page()` 函数始终都是从高地址端向低地址端遍历所有页面的，只要发现空闲页面，就申请，直到申请不到空闲页面为止。这可以保证所有申请到的页面在内存中都紧密排列，使得在4 GB的线性地址空间中分散的进程内存集约到有限的物理内存中运行。

当然，如果确实没有申请到空闲页面，说明此时内存中已经没有页面供进程使用了，所以就要直接返回错误信息，进程创建就此结束。系统

怠速后内存中有大量的空闲页面，str1进程又是怠速后刚刚创建的进程，所以此时可以申请到空闲页面。根据前面确定的task[64]中的项号，把该进程的task_struct挂接到task[64]中，task[64]的项数和线性地址空间的64等分布局正好相符。每创建一个进程，都要把task_struct的指针载入，这样系统如果查找进程，只要查task[64]，就可以顺着指针找到唯一的task_struct，不会出现混乱。代码如下：

```
//代码路径: kernel/fork.c:

int copy_process (int nr,long ebp,long edi,long esi,long gs,long
none, //这个nr就是task[64]中的项号

long ebx,long ecx,long edx,

long fs,long es,long ds,

long eip,long cs,long eflags,long esp,long ss)
```

```

{

struct task_struct * p;

int i;

struct file * f;

p= (struct task_struct *) get_free_page () ; //为str1进程申请空闲
页面

if (! p) //如果没有申请到空闲页面，返回错误信息

return-EAGAIN;

task[nr]=p; //将str1进程的task_struct挂接到task[64]中

*p=*current; /*NOTE ! this doesn't copy the supervisor stack*/

p->state=TASK_UNINTERRUPTIBLE;

p->pid=last_pid;

.....

}

```

此过程情景如图6-5所示。

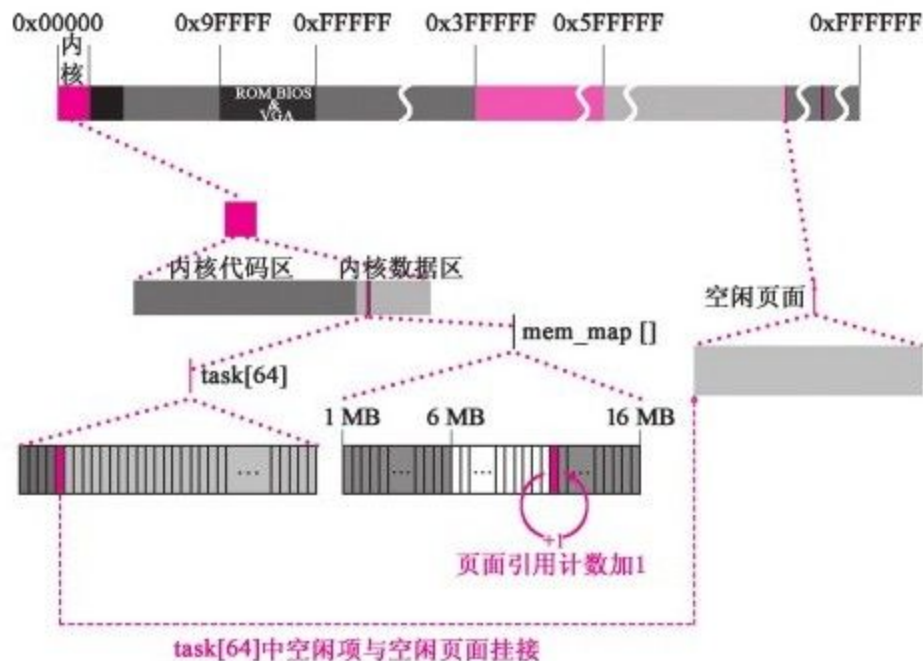


图 6-5 为str1进程task_struct申请页面并与操作系统挂接

3.Shell进程给str1进程复制task_struct结构

在Linux 0.11设计者看来，整个操作系统中只有进程，内核是进程的延续，所有的任务都应该由进程来承担。沿着这个设计思想不难发现，任何时候都要有一个进程在工作，current指针的意

思就是当前进程。创建一个进程，这个任务要由进程来完成，准确地说，要由当前进程来完成，由shell把自己的task_struct结构复制给str1进程，这也是该设计思想的延伸。执行代码如下：

//代码路径： kernel/fork.c:

```
int copy_process (int nr,long ebp,long edi,long esi,long gs,long  
none, //这个nr就是task[64]中的项号
```

```
long ebx,long ecx,long edx,
```

```
long fs,long es,long ds,
```

```
long eip,long cs,long eflags,long esp,long ss)
```

```
{
```

```
.....
```

```
if (! p)
```

```
return-EAGAIN;
```

```
task[nr]=p;
```



```
*p=*current; /*NOTE! this doesn't copy the supervisor stack*///复制  
task_struct结构给str1进程
```

```
p->state=TASK_UNINTERRUPTIBLE;
```

```
p->pid=last_pid;
```

```
.....
```

```
}
```

复制情景如图6-6所示。

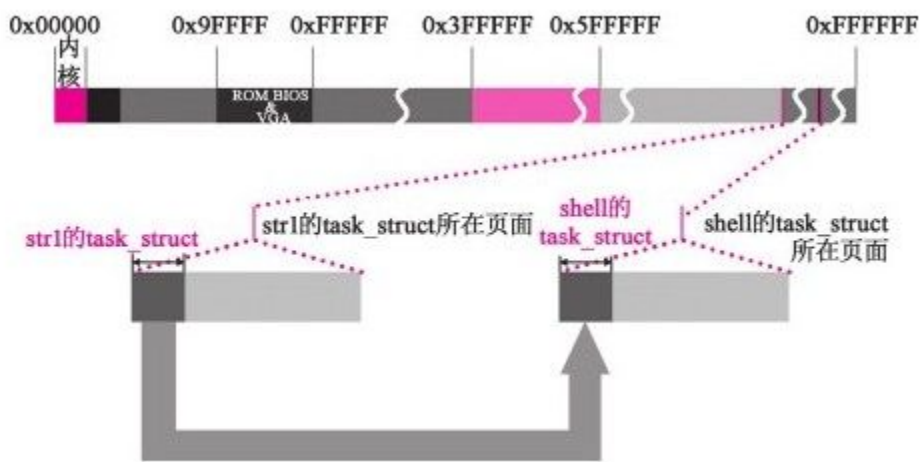


图 6-6 复制task_struct结构给str1进程

给str1复制了task_struct后，str1进程便继承了shell的全部管理信息。但由于每个进程的

task_struct结构中数据信息是不一样的，所以还要对该结构进行个性化设置。先将进程设置为不可中断等待状态。内核执行过程中，Linux 0.11不允许进程间切换，所以这里不进行状态设置也无所谓。但如果允许进程在内核执行时切换，这里就有必要设置为不可中断等待状态了。这是因为进程task_struct结构已经挂接在task[64]结构中，如果在个性化设置过程中发生时钟中断，就会轮转到该进程，而此时其个性化设置尚未完成，一旦切换到该进程去执行，就会引起进程执行的混乱。代码如下：

```
//代码路径: kernel/fork.c:

int copy_process (int nr,long ebp,long edi,long esi,long gs,long
none,

long ebx,long ecx,long edx,
```

```
long fs,long es,long ds,

long eip,long cs,long eflags,long esp,long ss)

{

.....

task[nr]=p;

*p=*current; /*NOTE ! this doesn't copy the supervisor stack*/

p->state=TASK_UNINTERRUPTIBLE; //str1进程被设置为不可中
断等待状态

p->pid=last_pid;

p->father=current->pid;

.....

}
```

这个过程如图6-7所示。

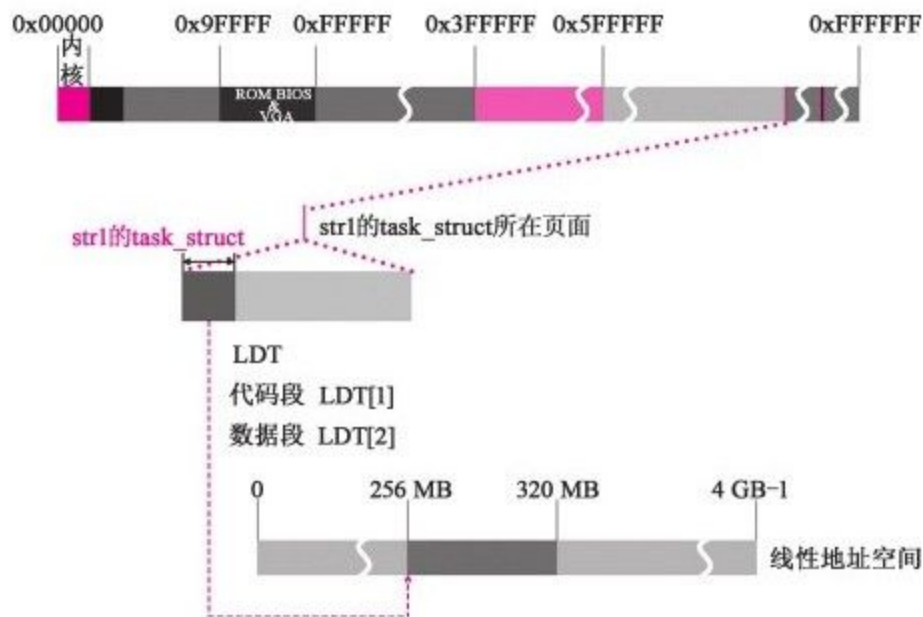


图 6-7 确定str1进程在线性空间中的位置

4.复制str1进程页表并设置其对应的页目录项

task_struct结构中还有其他字段需要个性化设置。str1进程的进程号和shell进程的进程号不一样；str1的父进程是shell，和shell进程的父进程也不一样。这些都需要进行个性化设置，代码如下：

//代码路径: kernel/fork.c:

```
int copy_process (int nr,long ebp,long edi,long esi,long gs,long
none,

long ebx,long ecx,long edx,

long fs,long es,long ds,

long eip,long cs,long eflags,long esp,long ss)

{

.....

*p=*current; /*NOTE! this doesn't copy the supervisor stack*/

p->state=TASK_UNINTERRUPTIBLE;

p->pid=last_pid; //设置str1进程的进程号

p->father=current->pid; //设置shell为str1进程的父进程

p->counter=p->priority;

p->signal=0;

.....

}
```

str1进程的时间片数值是从shell进程继承过来的。此时shell进程可能已经执行过一段时间了，时间片可能已经减少。但不能因为这些，就让str1进程的时间片也天然地减少，所以这里不能继承shell的时间片，而是用shell的优先级来确定时间片。如果优先级没有被用户指定设置过，它的数值和时间片的原始数值是一样的，即15。代码如下：

```
//代码路径: kernel/fork.c:

int copy_process (int nr,long ebp,long edi,long esi,long gs,long
none,

long ebx,long ecx,long edx,

long fs,long es,long ds,

long eip,long cs,long eflags,long esp,long ss)

{

.....
```

```
*p=*current; /*NOTE ! this doesn't copy the supervisor stack*/

p->state=TASK_UNINTERRUPTIBLE;

p->pid=last_pid;

p->father=current->pid;

p->counter=p->priority; //用当前进程的优先级设置str1进程的时间片

p->signal=0;

p->alarm=0;

.....

}
```

接下来是对信号的个性化设置。task_struct结构中，关于信号的字段有三个，即signal、sigaction[32]和blocked。它们分别表示信号位图、信号处理函数挂接点和信号屏蔽码。现在创建str1进程，只把signal进行了清零，其他的都没有设置。这样做是有原因的，如果不清零，那么就等

于默认沿用了父进程的信号位信息，将来str1执行时，如果执行到内核，则返回之前就要信号检测，本来str1没有接收到信号，就因为误用信号位信息，导致没必要的信号处理。为此需要改变用户栈信息、绑定进程的信号处理函数等一系列配合才能处理信号，而str1进程根本没准备这些，带着这些未知因素从内核返回进程，执行就彻底混乱了。

既然此时没有接收信号，那就没必要关系信号处理函数的挂接和屏蔽码了，所以其余两个字段不进行个性化设置，代码如下：

```
//代码路径： kernel/fork.c:

int copy_process (int nr,long ebp,long edi,long esi,long gs,long
none,

long ebx,long ecx,long edx,
```



```
long fs,long es,long ds,  
  
long eip,long cs,long eflags,long esp,long ss)  
  
{  
  
.....  
  
p->father=current->pid;  
  
p->counter=p->priority;  
  
p->signal=0; //信号位图清零  
  
p->alarm=0;  
  
p->leader=0; /*process leadership doesn't inherit*/  
  
.....  
  
}
```

信号的详细处理过程，我们将在第8章介绍。

下面我们继续介绍其他字段的个性化设置。
类似的清零和默认继承还表现在关于str1进程的时间设置方面和会话组织方面。代码如下：

//代码路径: kernel/fork.c:

```
int copy_process (int nr,long ebp,long edi,long esi,long gs,long
none,

long ebx,long ecx,long edx,

long fs,long es,long ds,

long eip,long cs,long eflags,long esp,long ss)

{

.....

p->counter=p->priority;

p->signal=0;

p->alarm=0; //报警定时值清零

p->leader=0; /*process leadership doesn't inherit*///会话首领字段
清零

p->utime=p->stime=0;

p->cutime=p->cstime=0;

p->start_time=jiffies;

p->tss.back_link=0;
```

```
p->tss.esp0=PAGE_SIZE+ (long) p;
```

```
.....
```

```
}
```

其他与时间设置和会话组织相关的字段全部沿用。以上这些字段是为内核管理进程而设置的。

下面介绍TSS字段。它是为进程间切换而设计的。进程切换是建立在对进程保护的基础上的。采取什么样的保护设计，就会有怎样的进程切换模式与之相适应。进程执行时，会用到各种寄存器。这导致进程切换不会是一个简单的跳转，而是一整套寄存器的值随之切换。这就要保证进程切换走时与切换回来时不发生混乱，这样才能保证进程执行的正确性。为此，Linux 0.11设

计者在每个进程的task_struct中记笔账，这笔账就是TSS。进程切换就要与此相适应。每当进程切换，就用TSS来保存现场，即保存当前各个寄存器的状态，等切换回这个进程后，再用TSS中的数据恢复寄存器中的数据。代码如下：

```
//代码路径: kernel/fork.c:

int copy_process (int nr,long ebp,long edi,long esi,long gs,long
none,

long ebx,long ecx,long edx,

long fs,long es,long ds,

long eip,long cs,long eflags,long esp,long ss)

{

.....

p->cutime=p->cstime=0;

p->start_time=jiffies;

p->tss.back_link=0; //以下代码是对TSS字段的设置
```

p->tss.esp0=PAGE_SIZE+ (long) p;

p->tss.ss0=0x10;

p->tss.eip=eip;

p->tss.eflags=eflags;

p->tss.eax=0;

p->tss.ecx=ecx;

p->tss.edx=edx;

p->tss.ebx=ebx;

p->tss.esp=esp;

p->tss.ebp=ebp;

p->tss.esi=esi;

p->tss.edi=edi;

p->tss.es=es&0xffff;

p->tss.cs=cs&0xffff;

p->tss.ss=ss&0xffff;

p->tss.ds=ds&0xffff;

p->tss.fs=fs&0xffff;

```
p->tss.gs=gs&0xffff;

p->tss.ldt=_LDT (nr) ; //以上代码是对TSS字段的设置

p->tss.trace_bitmap=0x80000000;

if (last_task_used_math==current)

__asm__ ("clts; fnsave%0": "m" (p->tss.i387) ) ;

.....

}
```

从以上代码中可以看出，`copy_process ()` 函数参数中设置的变量，基本都用来设置段寄存器；将来`str1`开始执行的契机，也必将是一次进程切换导致的；而一旦发生进程切换，**TSS**这一整套的寄存器值都伴随过去，用来初始化CPU的各个寄存器，并决定`str1`进程开始的执行状态。

另外值得注意的是，用这些数值来设置寄存器，是CPU自动完成的，所以我们在内核中找不到给寄存器赋值的代码。那么CPU中的硬件怎么知道哪个数值是用来为哪个寄存器赋值的呢？只有一种可能，就是CPU硬件的事先约定，按照目前TSS中各个字段的顺序取值。这意味着，如果顺序排列错误，进程执行就混乱了。

对进程的保护不仅体现在内核对进程的管理以及进程切换时的控制，在进程执行的过程中，也要有一整套措施时刻看着进程的边界，具体表现为分段和分页。下面我们继续介绍str1分段和分页的情况。

5.复制str1进程页表并设置其对应的页目录项

现在调用`copy_mem`（）函数为进程分段，即确定线性地址空间。

通过前面的介绍我们得知，确定线性地址空间，关键在于确定段基址和段限长。执行代码如下：

//代码路径：kernel/fork.c:

```
int copy_mem (int nr,struct task_struct * p)
{
    unsigned long old_data_base,new_data_base,data_limit;

    unsigned long old_code_base,new_code_base,code_limit;

    code_limit=get_limit (0x0f) ; //获取当前进程，即shell代码段段
    限长

    data_limit=get_limit (0x17) ; //获取当前进程，即shell数据段段
    限长

    old_code_base=get_base (current->ldt[1]) ;

    old_data_base=get_base (current->ldt[2]) ;
```



```
if (old_data_base != old_code_base)
```

```
panic ("We don't support separate I&D") ;
```

```
if (data_limit < code_limit)
```

```
panic ("Bad data_limit") ;
```

new_data_base=new_code_base=nr*0x4000000; //根据在task[64]中
确定的项号nr, 确定段基址

```
p->start_code=new_code_base;
```

set_base (p->ldt[1], new_code_base) ; //参考str1进程代码基址
设置它的LDT

set_base (p->ldt[2], new_data_base) ; //参考str1进程数据段基
址设置它的LDT

```
if (copy_page_tables (old_data_base,new_data_base,data_limit) )  
{
```

```
free_page_tables (new_data_base,data_limit) ;
```

```
return-ENOMEM;
```

```
}
```

```
return 0;
```

```
}
```

值得注意的是，从以上代码中我们看到了根据task[64]中的项号nr来确定str1进程段基址，并参考段基址设置了str1进程的LDT，但始终没有看到设置它的段限长。通过前面的介绍我们得知，进程的段限长存储在LDT中，复制task_struct结构时，把LDT也复制过来了，没有改变。这说明str1沿用了其父进程shell的LDT。这样做的理由是，str1进程开始执行后总要执行代码，但它还没有加载属于自己的程序（或许以后也不加载了），这样就只能和父进程共用代码。此时沿用父进程的段限长，就是为了能够共享到父进程的全部代码和数据。

分段问题处理完毕后，下面开始分页。分页是建立在分段基础上的，具体表现为，分段时用

段基址和段限长分别为分页确定了从哪里开始复制页面表项信息、复制到哪里去和复制多少这三件事，代码如下：

```
//代码路径: kernel/fork.c:

int copy_mem (int nr,struct task_struct * p)

{

.....

new _data_base=new_code_base=nr*0x4000000;

p->start_code=new_code_base;

set_base (p->ldt[1], new_code_base) ;

set_base (p->ldt[2], new_data_base) ;

if (copy_page_tables (old_data_base,new_data_base,data_limit) )
{//调用此函数为str1进程分页

free _page_tables (new_data_base,data_limit) ;

return-ENOMEM;

}
```

```
return 0;
```

```
}
```

前面分段时我们介绍到，str1创建后，还没有自己的程序，还要和父进程shell共享程序。这一点在分页时表现为与shell进程共享页面，即为str1进程另起一套页目录项和页表项，使之和shell指向共同的页面。代码如下：

```
//代码路径： mm/memory.c:
```

```
int copy_page_tables (unsigned long from,unsigned long to,long  
size)
```

```
{
```

```
.....
```

```
for (; size-->0; from_dir++, to_dir++) { //这里体现了分段是分  
页的基础
```

```
if (1&*to_dir)
```

```
panic ("copy_page_tables: already exist") ;
```

```

if ( ! (1 & *from_dir) )

continue;

from_page_table = (unsigned long *) (0xffff000 & *from_dir) ;

if ( ! (to_page_table = (unsigned long *) get_free_page ( ) ) ) //
为创建页表新申请一个页面

return -1; /*Out of memory, see freeing*/

*to_dir = ( (unsigned long) to_page_table ) | 7; //设置页目录项

nr = (from == 0) ? 0xA0 : 1024;

for ( ; nr-- > 0; from_page_table++, to_page_table++) { //复制页
表

this_page = *from_page_table;

if ( ! (1 & this_page) )

continue;

this_page &= ~2; //这里的设置使共享的页面对于shell进程来讲
是只读的

*to_page_table = this_page; //这里的设置使共享的页面对于str1进
程来讲是只读的

if (this_page > LOW_MEM) {

*from_page_table = this_page;

```

```

this_page-=LOW_MEM;

this_page>>=12;

mem_map[this_page]++;

}

}

}

invalidate ();

return 0;

}

```

复制页表和设置页目录项的过程如图6-8所示。

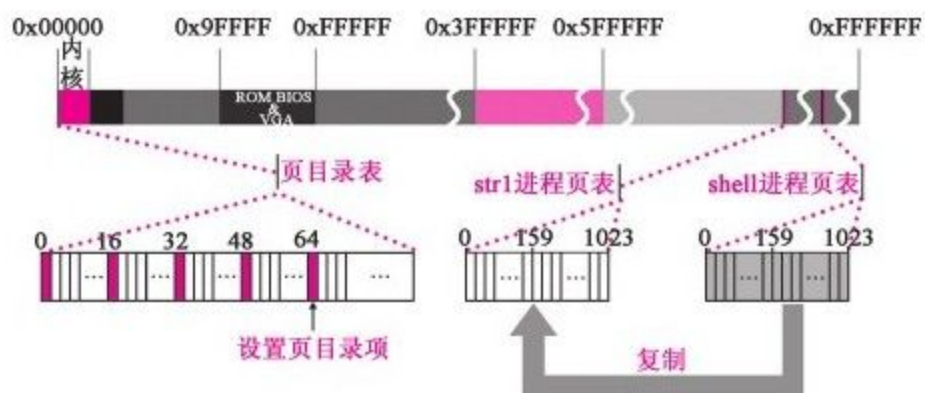


图 6-8 为str1进程复制页表并设置str1进程的页
目录项

值得注意的是，为新进程创建页表时，还要调用`get_free_page()`函数申请空闲页面。从需求上来看，这里申请的页面，将用来承载新进程的页表项。这些页表项是用来管理str1所占用的页面的，是不让进程使用的，所以这里只申请了页面，并没有映射到str1进程的线性地址空间内。与前面为进程的`task_struct`和内核栈申请页面类似，这里申请页面时，程序是在内核中执行的，处于内核的线性地址空间内，此时申请的页面早已映射到内核的线性地址空间内，内核因此具备访问它的能力。同理，现在为装载页表而申请的页面，都是内核用来管理进程的，内核能访问到，

进程访问不到。进程访问不到的根本原因是，内核没有把这些页面映射到进程的线性地址空间内。

分段、分页完成后，还有文件继承的问题要处理。shell进程打开的文件，它的子进程一并继承，具体表现为，将文件的引用计数和i节点的引用计数累加，将来子进程需要使用这些文件时，就可以直接操作，不需要再重新打开。比如说，shell进程怠速前打开了tty文件，还复制了文件句柄，str1进程就可以直接用，不需要重新读取了。引用计数累加的执行代码如下：

```
//代码路径: kernel/fork.c:

int copy_process (int nr,long ebp,long edi,long esi,long gs,long
none,

long ebx,long ecx,long edx,
```



```

long fs,long es,long ds,

long eip,long cs,long eflags,long esp,long ss)

{

.....

if (copy_mem (nr,p) ) {

task[nr]=NULL;

free _page ( (long) p) ;

return-EAGAIN;

}

for (i=0; i<NR_OPEN; i++)

if (f=p->filp[i])

f->f_count++; //累加文件引用计数

if (current->pwd)

current->pwd->i_count++; //累加当前工作目录i节点引用计数

if (current->root)

current->root->i_count++; //累加当前根目录i节点引用计数

if (current->executable)

```

```
current->executable->i_count++; //累加可执行文件i节点的引用  
计数
```

```
set_tss_desc (gdt+ (nr<<1) +FIRST_TSS_ENTRY, & (p->  
tss) ) ;
```

```
set_ldt_desc (gdt+ (nr<<1) +FIRST_LDT_ENTRY, & (p->  
ldt) ) ;
```

```
.....
```

```
}
```

6.建立str1进程与全局描述符表（GDT）的关联

文件继承问题解决后，将str1的进程TSS和LDT挂接在GDT的指定位置处。代码如下：

```
//代码路径： kernel/fork.c:
```

```
int copy_process (int nr,long ebp,long edi,long esi,long gs,long  
none,
```

```
long ebx,long ecx,long edx,
```

```

long fs,long es,long ds,

long eip,long cs,long eflags,long esp,long ss)

{

.....

if (current->pwd)

current->pwd->i_count++;

if (current->root)

current->root->i_count++;

if (current->executable)

current->executable->i_count++;

set_tss_desc (gdt+ (nr<<1) +FIRST_TSS_ENTRY, & (p->
tss) ) ; //将str1进程的TSS挂接在GDT上并设置段信息

set_ldt_desc (gdt+ (nr<<1) +FIRST_LDT_ENTRY, & (p->
ldt) ) ; //将str1进程的LDT挂接在GDT上并设置段信息

p->state=TASK_RUNNING; /*do this last,just in case*/

return last_pid;

}

```

设置过程如图6-9所示。

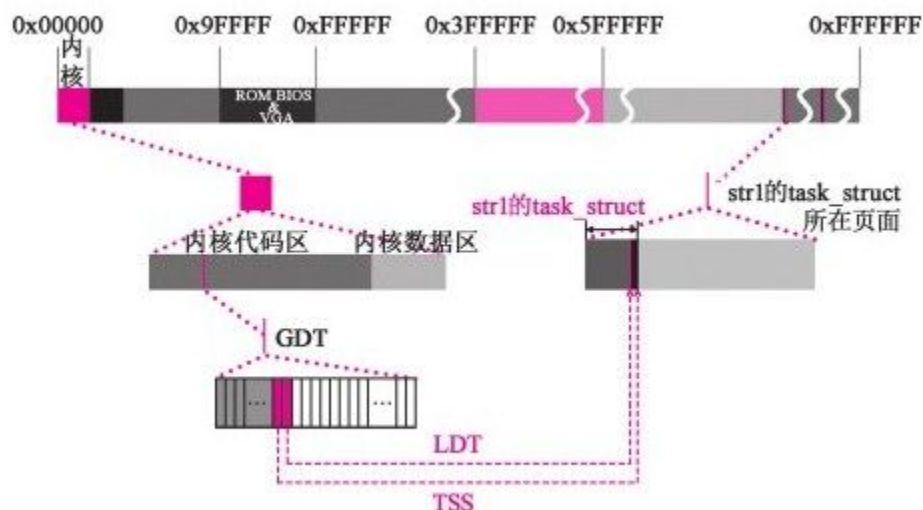


图 6-9 str1进程的TSS和LDT与GDT挂接

TSS和LDT对进程的保护至关重要。保护的本质，就是不能让进程执行时干扰到其他进程。在“段”这方面，有以下两种方式可以跳到其他进程。

第一，前面讲解过，执行段内跳转指令，但跳转值超过了段限长。这一点硬件始终在关注，

LDT中记录了进程的段基址和段限长，每执行一条指令，硬件都会检查是否超过设定的范围，如果超过了，就直接报GP了，强行拦截。

第二，进程执行时，执行段间跳转指令，以此实现段间跳转。Linux 0.11中，每个进程一套LDT，此时进程是在3特权级下执行的，用LDT中提供的段描述符。在这种情况下，如果要实现段间跳转，要把当前的LDT更换为其他段的LDT，以此更改段描述符，就需要LDTR中LDT的基址，执行的指令是“LLDT”，而这条指令只能在0特权级下执行，所以此时无法改变LDT。这就导致，进程无论怎么做，都无法跳转到其他段，也就是无法直接跳转到其他进程。

假设Linux 0.11的保护格局不是这样的，所有进程都不用LDT来记录段描述符，而是统一用GDT，那就不用执行LGDT指令来改变GDT了，就可以实现段间跳转，打破了保护。可见，设计者对段一级的保护很是费了一些心思。

这两条路都堵死了，段一级的保护就算完善了。在段一级保护实现的基础上，再实现页一级实施保护。从前面我们对两次get_free_page ()函数的调用可知，分页动作完全都是由内核完成的。如果内核不把页面映射到进程的线性地址空间内，进程是无法访问页面的。如果映射，就要指定页目录表和页表，而页目录表中页目录项的确定，完全取决于进程的线性地址空间，只要线

性地址空间不重叠，分页肯定不会引起进程内存的混乱。

7.将str1进程设为就绪态

创建str1进程的过程到此结束了，将它的状态设置为“就绪态”。这意味着str1进程可以参与轮转了。

执行代码如下：

```
//代码路径: kernel/fork.c:

int copy_process (int nr,long ebp,long edi,long esi,long gs,long
none,

long ebx,long ecx,long edx,

long fs,long es,long ds,

long eip,long cs,long eflags,long esp,long ss)

{
```

```

.....

set_tss_desc (gdt+ (nr<<1) +FIRST_TSS_ENTRY, & (p->
tss) ) ;

set_ldt_desc (gdt+ (nr<<1) +FIRST_LDT_ENTRY, & (p->
ldt) ) ;

p->state=TASK_RUNNING; /*do this last,just in case*///设置为就
绪态

return last_pid;

}

```

这一过程如图6-10所示，其中的state被设置为了TASK_RUNNING。

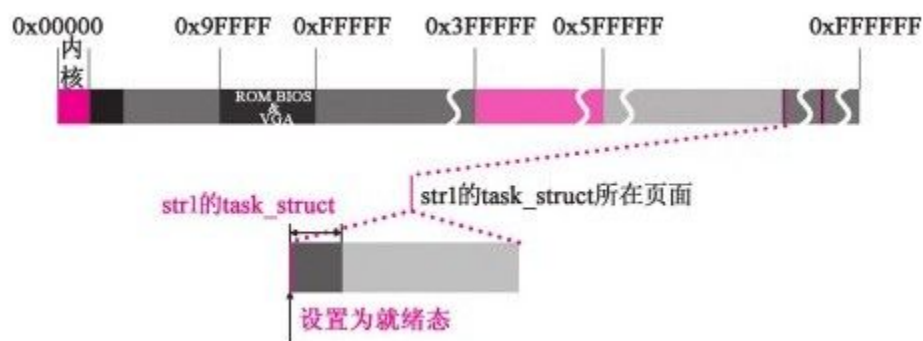


图 6-10 将str1设置为就绪态

6.3.2 str1进程加载的准备工作

1.为用户进程str1的加载做准备

为用户进程str1的加载做准备与为shell程序的加载做准备的方式是大体相同的，包括对参数和环境变量等外围环境的检测、对str1可执行文件的检测、对str1进程task_struct的针对性调整以及最终设置EIP、ESP这几部分。

进入do_execve函数后，先要做外围准备工作，即为管理str1进程参数和环境变量所占用的页面做准备，还需要把str1所在的文件i节点读出来，通过i节点信息，检测文件自身是否有问题，再通过i节点找到文件头，对文件进行检测，其中包括检查记录的可执行文件代码长度、数据长

度，能不能容纳在64 MB的线性地址空间内。代码如下：

```
//代码路径: fs/exec.c:

int do_execve (unsigned long * eip,long tmp,char * filename,
char ** argv,char ** envp)
{
.....

if (N_MAGIC (ex) !=ZMAGIC||ex.a_trsize||ex.a_drsz||
ex.a_text+ex.a_data+ex.a_bss> 0x3000000||//代码、数据、堆的总长度不能超过48 MB

inode->i_size< ex.a_text+ex.a_data+ex.a_syms+N_TXTOFF
(ex) ) {

retval=-ENOEXEC;

goto exec_error2; //如果超过48 MB，直接跳转到错误处

}

.....

}
```

这些检查完成，标志着str1程序在文件格式的规范性方面没有问题，而且能够容纳在64 MB的线性地址空间内。有了这个前提，再往下调整才有意义。

创建str1进程时我们介绍到，它共享了一些shell进程打开的文件，继承了一些shell进程的信号字段内容。现在它要加载自己的程序了，所以有些关系要解除，有些要清零。代码如下：

//代码路径：fs/exec.c:

```
int do_execve (unsigned long * eip,long tmp,char * filename,
char ** argv,char ** envp)
{
.....
if (! sh_bang) {
```

```
p=copy_strings (envc,envp,page,p, 0) ;
```

```
p=copy_strings (argc,argv,page,p, 0) ;
```

```
if (! p) {
```

```
    retval=-ENOMEM;
```

```
    goto exec_error2;
```

```
}
```

```
}
```

```
/*OK,This is the point of no return*/
```

if (current->executable) //要从str1这个可执行文件中载入程序了，就不需要共享shell进程可执行文件的i节点了

iput (current->executable) ; //解除与shell进程可执行文件i节点的关系

```
current->executable=inode; //str1可执行文件的i节点取代
```

```
for (i=0; i<32; i++)
```

current->sigaction[i].sa_handler=NULL; //把信号句柄清空，准备加载用户自己的信号处理程序

```
for (i=0; i<NR_OPEN; i++)
```

```
if ( (current->close_on_exec > i) & 1)
```

```
sys_close (i) ;
```

```
current->close_on_exec=0; //把打开文件的屏蔽位清零

    free_page_tables (get_base (current->ldt[1]), get_limit
(0x0f) ) ;

    free_page_tables (get_base (current->ldt[2]), get_limit
(0x17) ) ; .....

}
```

2.释放str1进程的页表

前面已经介绍str1进程现在与shell进程正在共享着相同的页面，现在str1要加载自己的程序了，需要解除共享关系，通过调用free_page_tables

() 函数来实现。执行代码如下：

//代码路径： fs/exec.c:

```
int do_execve (unsigned long * eip,long tmp,char * filename,
char ** argv,char ** envp)
{
```

```

.....

for (i=0; i<NR_OPEN; i++)

if ( (current->close_on_exec>>i) &1)

sys _close (i) ;

current->close_on_exec=0;

free _page_tables (get_base (current->ldt[1]) , get_limit
(0x0f) ) ; //释放代码段共享的页面

free _page_tables (get_base (current->ldt[2]) , get_limit
(0x17) ) ; //释放数据段共享的页面

if (last_task_used_math==current)

last_task_used_math=NULL;

current->used_math=0;

.....

}

//代码路径: mm/memory.c:

int free_page_tables (unsigned long from,unsigned long size)

{

.....

```

```

if (1 & *pg_table)

free_page (0xfffff000 & *pg_table) ; //释放掉共享的页面

*pg_table=0; //页表项清零

pg_table++;

}

free_page (0xfffff000 & *dir) ; //释放掉页表自身占用的页面

*dir=0; //页目录项清零

.....

}

```

释放页表的过程如图6-11所示，请读者注意其中页目录项的变化。

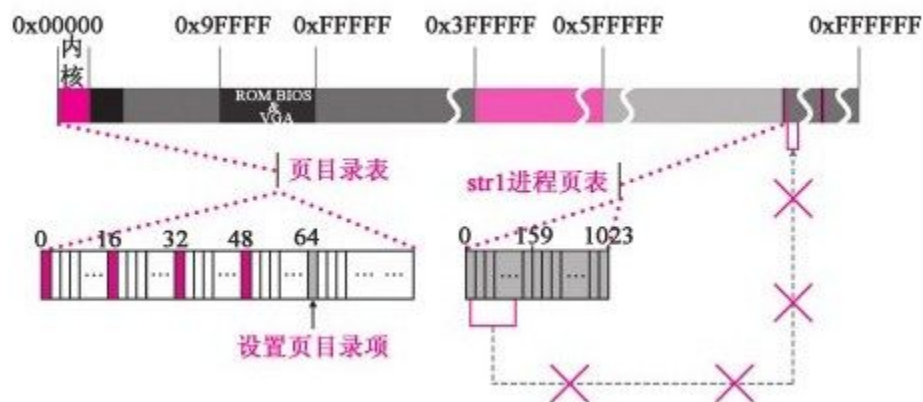


图 6-11 释放str1进程继承的页面

值得注意的是，原来str1进程与shell进程是共享页面的，即对于这两个进程，共享的页面都是只读的。现在解除的是str1和共享页面的关系，但这些页面对于shell进程来讲仍然是只读的，那么这样会不会影响shell进程将来的执行？这一点我们在后面讲解写时复制时会详细介绍。

3.重新设置str1的程序代码段和数据段

str1进程要加载自己的程序，需要根据程序的长度来重新设置LDT，代码如下：

```
//代码路径: fs/exec.c:
```

```
int do_execve (unsigned long * eip,long tmp,char * filename,  
char ** argv,char ** envp)
```



```

{

.....

    p+=change_ldt (ex.a_text,page) -MAX_ARG_PAGES *
PAGE_SIZE; //重新设置段限长

    .....

}

static unsigned long change_ldt (unsigned long text_size,unsigned
long * page)

{

.....

    code_limit=text_size+PAGE_SIZE-1;

    code_limit&=0xFFFFF000; //根据代码长度重新设置了代码段的
段限长

    data_limit=0x4000000; //将数据段长度设置为64 MB

    code_base=get_base (current->ldt[1]) ;

    data_base=code_base; //段基址没有变化

    set_base (current->ldt[1], code_base) ;

    set_limit (current->ldt[1], code_limit) ;

    set_base (current->ldt[2], data_base) ;

```

```
set_limit (current->ldt[2], data_limit) ;  
  
.....  
  
}
```

这里是对str1进程段限长的最后设置，可以看出，始终没有超过64 MB。如果str1进程以后做父进程，根据进程复制机制，它创建出来的子进程的段限长也不会超过64 MB，这样，系统创建的每个进程，都被限制在属于自己的64 MB地址空间内。

设置过程如图6-12所示。

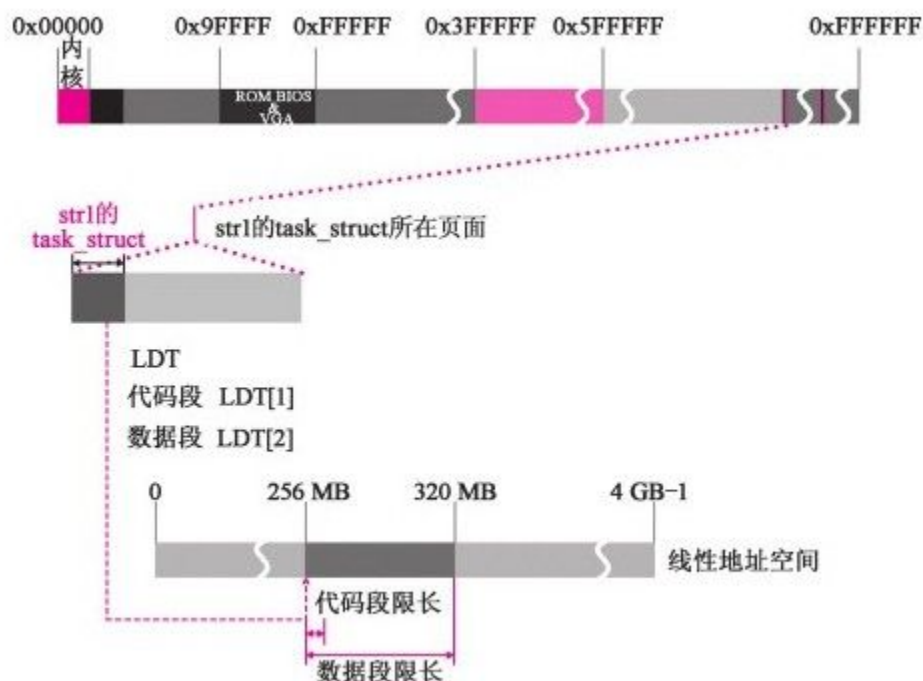


图 6-12 重新设置str1程序的代码段和数据段

4.调整str1进程task_struct

对str1进程task_struct中的brk、start_stack等信息进行设置。设置这些字段的作用，是为了避免进程在执行时发生错误，本质上是管理，不是保护。执行代码如下：

```
//代码路径: fs/exec.c:
```

```

int do_execve (unsigned long * eip,long tmp,char * filename,

char ** argv,char ** envp)

{

.....

current->used_math=0;

p+=change_ldt (ex.a_text,page) -MAX_ARG_PAGES *
PAGE_SIZE;

p= (unsigned long) create_tables ( (char *) p,argc,envc) ;

current->brk=ex.a_bss+//以下是根据文件头ex中提供的信息，设置
程序控制字段

(current->end_data=ex.a_data+

(current->end_code=ex.a_text) ) ;

current->start_stack=p&0xfffff000;

current->euid=e_uid;

current->egid=e_gid;

i=ex.a_text+ex.a_data;

while (i&0xfff)

put_fs_byte (0, (char *) (i++) ) ;

```

```

eip[0]=ex.a_entry; /*eip,magic happens: -) */

eip[3]=p;

.....

}

```

设置结果装载到str1进程task_struct所占页面的过程如图6-13所示。

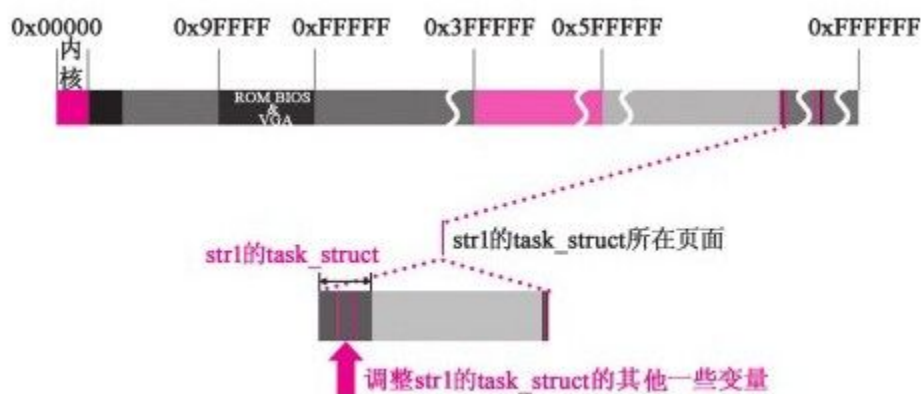


图 6-13 调整str1进程task_struct

最后再调整EIP和ESP，使软中断返回后，直接从str1程序开始位置执行。前面我们已经介绍过，str1进程与shell进程解除了共享页面的关系，

控制页面的页表也已经释放，断绝了与str1进程的页目录项的映射关系。这意味着页目录项的内容为0，包括P位也为0。str1程序一开始执行，MMU解析线性地址值时就会发现对应的页目录项P位为0，因此产生缺页中断。

6.3.3 str1进程的运行、加载

1.产生缺页中断并由操作系统响应

缺页中断信号产生后，`page_fault`这个服务程序将对此进行响应，并最终在`_page_fault`中通过“`call_do_no_page`”调用到缺页中断处理程序`_do_no_page`中去执行。

执行代码如下：

```
//代码路径: mm/page.s:
```

```
_page_fault:
```

```
.....
```

```
testl $1, %eax
```

```
jne 1f
```

```
1: call_do_no_page
```

```
.....
```

进入do_no_page（）函数后，在加载str1程序之前，先要做如下两方面的检测。

第一，str1进程是否已经把程序加载进来了，或者产生缺页中断的线性地址值是否已经超出了程序代码的末端。显然两种情况都不成立，str1的代码内容将从硬盘上加载。执行代码如下：

```
//代码路径：mm/memory.c:
```

```
void do_no_page（unsigned long error_code,unsigned long address）
```

```
{
```

```
.....
```

```
address&=0xfffff000;
```

```
tmp=address-current->start_code;
```



```

        if (! current->executable||tmp>=current->end_data)
{
    //executable为str1程序所在文件i节点， end_data为程序代码末端

    get_empty_page (address) ;

    return;

}

if (share_page (tmp) )

return;

.....

}

```

第二，str1是否有可能与某个现有的进程共享代码，比如某个其他进程已经把str1程序加载了的情况。显然现在也不可能。代码如下：

//代码路径： mm/memory.c:

```

void do_no_page (unsigned long error_code,unsigned long address)

{

```

```

.....

if (! current->executable||tmp>=current->end_data) {

get_empty_page (address) ;

return;

}

if (share_page (tmp) ) //试图和其他进程共享

return;

if (! (page=get_free_page () ) )

oom () ;

.....

}

```

现在的情况和加载shell程序时的情况一样，都需要从硬盘上把程序加载进来。下面就要在主内存中申请空闲页面，然后加载str1程序。

2.为str1程序申请一个内存页面

在主内存中申请一个空闲页面，准备将str1最起始部分的程序载入，执行代码如下：

//代码路径： mm/memory.c:

```
void do_no_page (unsigned long error_code,unsigned long address)
{
.....

if (share_page (tmp) )

return;

if ( ! (page=get_free_page ( ) ) ) //为str1申请页面

oom ( ) ; //要是申请不到空闲页面，就令str1进程退出

/*remember that 1 block is used for header*/

block=1+tmp/BLOCK_SIZE;

.....

}
```

为str1程序申请的空闲页面及其在内存管理结构mem_map中登记的情况如图6-14所示。

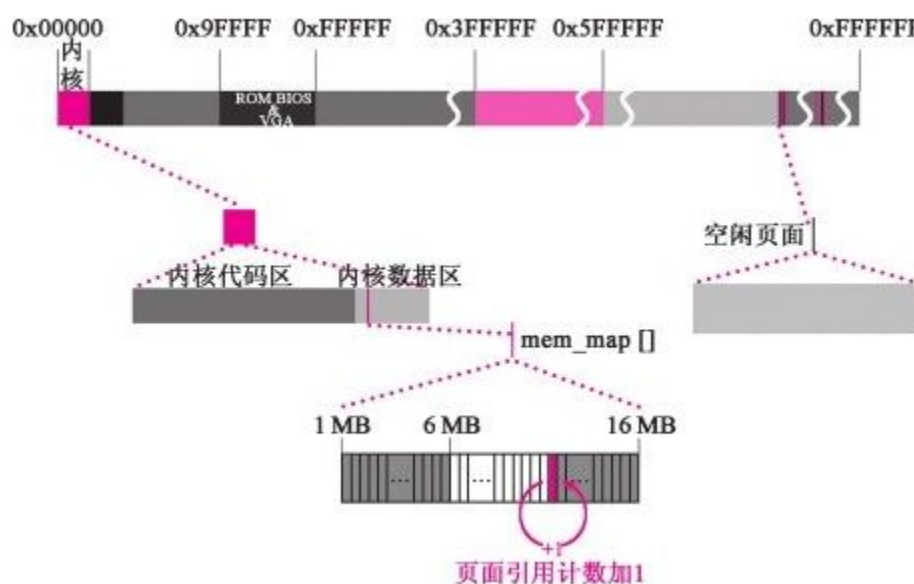


图 6-14 为str1程序申请一个内存页面

从前面的讲解可知，所有分配给进程的页面天然存在两套映射关系，一套是与内核线性地址空间的映射，另一套是与进程线性地址空间的映射。内核与页面的映射关系从来没被解除过。试想如果把页面映射给进程后就解除了内核与页面

的映射关系，那就意味着内核无法访问到这个页面了。

3.将str1程序加载到新分配的页面中

将str1程序从硬盘加载到这个新分配的页面中，一次加载4 KB的内容。执行代码如下：

//代码路径: mm/memory.c:

```
void do_no_page (unsigned long error_code,unsigned long address)
{
    .....

    if (! (page=get_free_page () ) )

        oom () ;

    /*remember that 1 block is used for header*/

    block=1+tmp/BLOCK_SIZE;

    for (i=0; i<4; block++, i++)
```

```
nr[i]=bmap (current->executable,block) ;

bread_page (page,current->executable->i_dev,nr) ; //从硬盘上
读取str1的信息

i=tmp+4096-current->end_data;

tmp=page+4096;

.....

}
```

其中，bmap（）函数在第5章5.5节已经详细讲解过；bread_page（）函数的执行过程本质上与bread（）函数相同。

过程如图6-15所示。

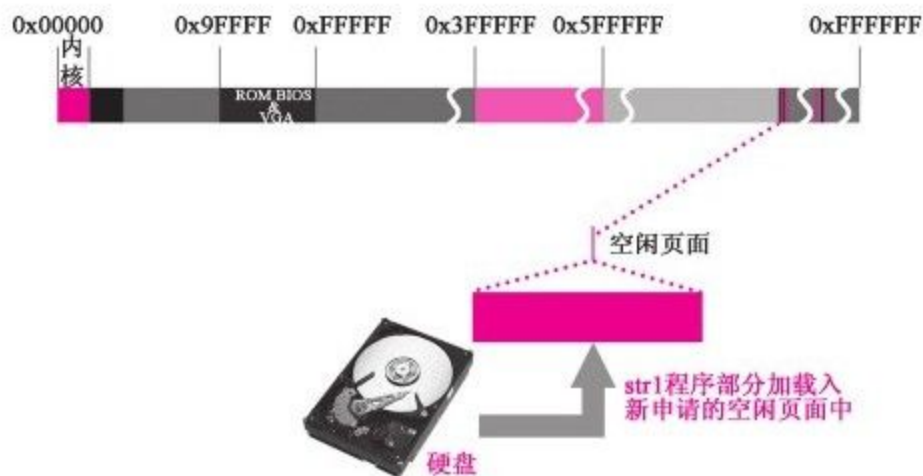


图 6-15 将str1程序的初始部分加载到新分配页面

由于这个页面早已经映射到了内核的线性地址空间内，所以新加载进来的内容，只要内核有需要，就可以进行任意改动，以后加载进来的内容也一样。

4.将分配给str1程序的物理内存地址与str1进程的线性地址空间对应

str1程序载入后，将这页内存映射到str1进程的线性地址空间内。对应的代码如下：

//代码路径： mm/memory.c:

```
void do_no_page (unsigned long error_code,unsigned long address)
{
    .....

    while (i-->0) {

        tmp--;

        * (char *) tmp=0;

    }

    if (put_page (page,address) ) //映射到str1的线性地址空间内

    return;

    free _page (page) ;

    oom ( ) ;

}
```

映射过程如图6-16所示。请读者注意在此过程中，对页目录项进行了设置。

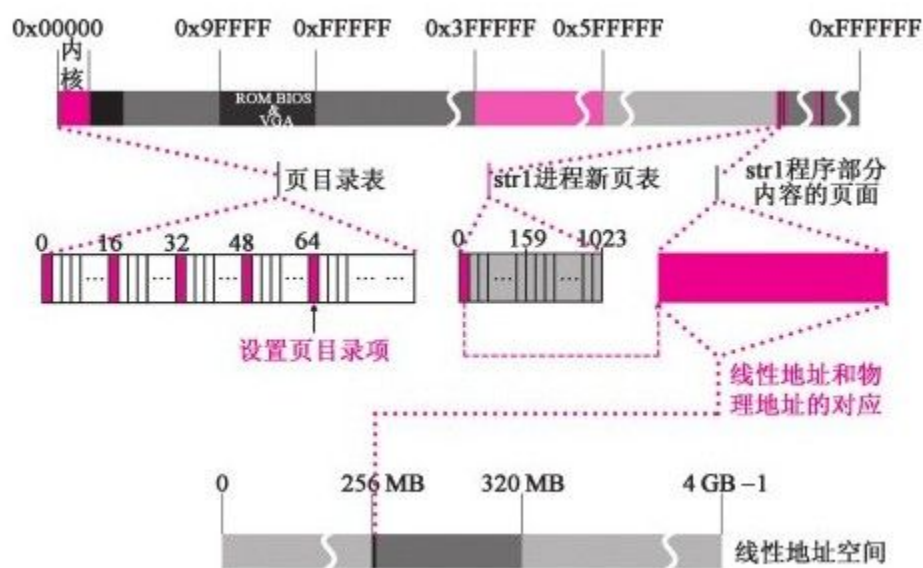


图 6-16 将str1程序的物理地址与线性地址对应

映射完毕后，str1进程才能执行到加载的程序。

5.不断通过缺页中断加载str1程序的全部内容

这个str1程序是大于一个页面的，所以，在执行过程中，如果需要新的代码，就会不断地产生缺页中断，以此来不断地加载需要执行的程序。

到这里为止，str1的加载过程就介绍完了。下面开始介绍str1运行起来的情况。

1.str1程序需要压栈

程序开始运行了，压栈动作产生。

str1程序中的foo函数被递归调用了，在foo函数中使用一个大小为2048的字符数组text（为了让缺页中断的效果更快地表现出来，所以我们这里设置的数组大小为2048字节，这样，两次压栈，效果就出来了），以使str1进程的栈空间以较

快的速度增长。每调用一次foo函数，str1进程栈（ESP）就向下增长2048字节。

2.str1程序第一次调用foo程序压栈

第一次调用foo函数，ESP向下扩充了2048字节。在扩充之前，ESP所指向的用户进程参数及环境变量列表已经在页面中占据了一定的空间。此时扩充2048字节，再加上原来参数列表占用的这部分空间，仍然没有超出一个页面的总空间4KB，所以，目前尚可以与列表共存于同一物理页面中，情况如图6-17所示。

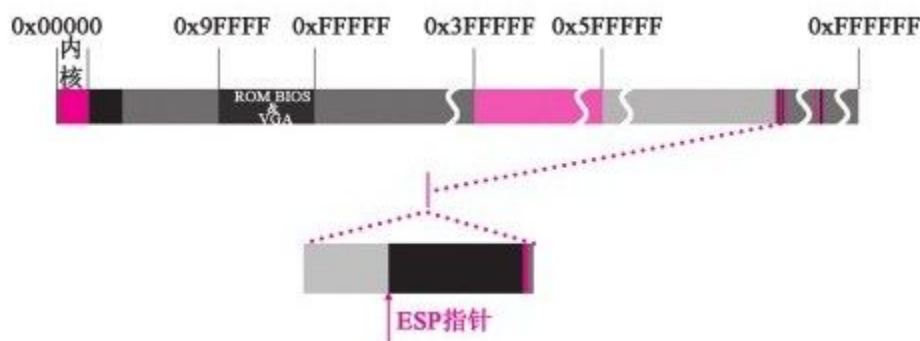


图 6-17 str1程序第一次压栈

3.str1程序第二次压栈，产生缺页中断

第二次调用foo函数时，情况就不同了。加上第一次扩充的2048字节，此时这个页面已经无法继续承载2048字节的内容了。MMU解析线性地址值的时候，会发现新的页表项中P位为0，因此再次产生缺页中断，准备申请新的页面。

4.处理str1程序第二次压栈产生的缺页中断

新的物理页面最终会映射到str1进程的线性地址空间内，以此支持寻址。这次缺页中断仍然是进入do_no_page（）函数去执行，但是执行的代码不同了，接下来会执行如下的代码：

//代码路径: mm/memory.c:

```
void do_no_page (unsigned long error_code,unsigned long address)
```

```
{
```

```
.....
```

```
address&=0xfffff000;
```

```
tmp=address-current->start_code;
```

```
if (! current->executable||tmp>=current->end_data) {//此条件为真
```

```
get_empty_page (address) ; //压栈时申请空闲页面
```

```
return;
```

```
}
```

```
.....
```

```
}
```

这是因为`tmp >= current->end_data`条件成立了, 此时程序在线性地址空间中执行到的线性地址值已经大于进程的`end_data`地址。因此, 接下

来调用`get_empty_page`（）函数去执行。这次并不会将任何外设的数据加载入页面，压栈也会产生缺页中断，但跟外设上的数据毫无关系。

进入`get_empty_page`（）函数后，就会新申请缺少的页面，并将这个物理页面映射到`str1`进程的线性地址空间内。执行代码如下：

```
//代码路径： mm/memory.c:

void get_empty_page (unsigned long address)

{

    unsigned long tmp;

    if (! (tmp=get_free_page () ) ||! put_page (tmp,address) ) {//
申请页面及映射到str1的线性地址空间内

        free _page (tmp) ; /*0 is ok-ignored*/

        oom () ;

    }
```

}

5.str1程序继续执行，反复压栈并产生缺页中断

程序继续执行，如此反复进行“压栈→如果表项P位为0→缺页中断→分配物理内存→压栈……”操作。当第n次调用foo函数时，用户进程栈与物理内存的映射关系如图6-18所示。请大家注意压栈数据在内存页面上的变化情况。

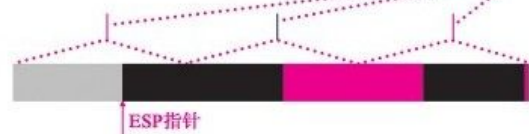
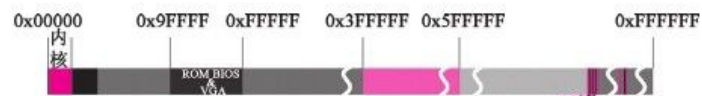
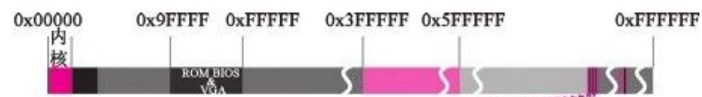
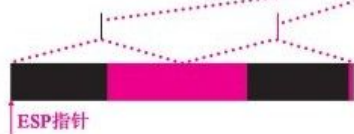
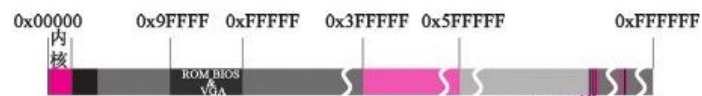
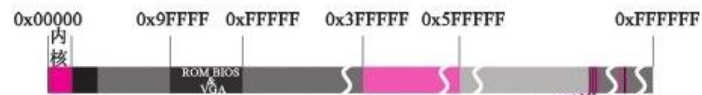
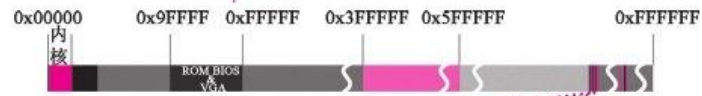
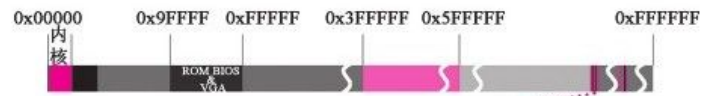


图 6-18 str1程序不断压栈的效果

6.str1程序运行结束后清栈

程序执行完毕，foo函数到达递归的终止点返回（if（n==0）return 0）。这时由于函数返回导致进程清栈，ESP向上（高地址）收缩，用户进程实际使用的栈空间就变小了。既然栈已经变小，那么之前映射到栈的线性地址空间的物理页面就应该被释放掉。但从代码分析和测试中都发现，Linux 0.11内核并没有释放该物理页面。理由是这样的：进程执行的时候内核不在执行状态，进程执行过程中废弃的页面，内核无法时时检测。而CPU中没有专门的功能电路来管理此事，没有判断废弃页面的硬件触发机制，内核即

使设计了清理废弃页面的功能，也无用武之地。
所以清栈后的页面没有被释放。

结果如图6-19所示。

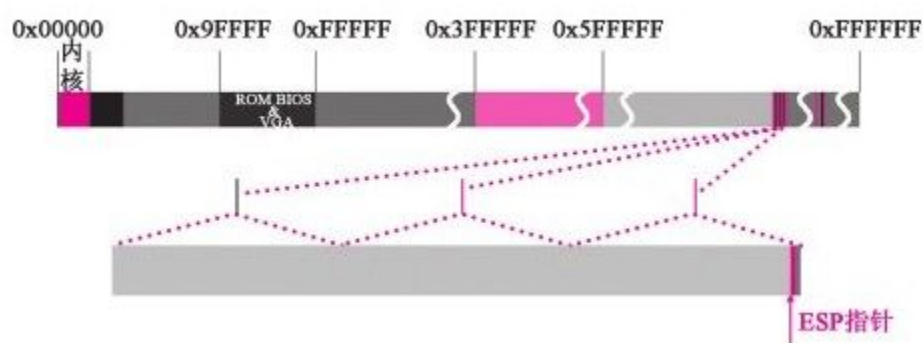


图 6-19 str1程序运行结束后清栈效果示意图

6.3.4 str1进程的退出

这里介绍str1进程的退出，包括运行过程中占据的内存空间如何释放，它自身task_struct所占据的空间如何处理，等等。其实str1的退出与shell进程的退出大体一致，都是通过调用exit（）函数来实现的，善后事务处理完毕后，由父进程负责将它的task_struct所在的页面释放掉。下面我们来具体介绍这一过程。

1.str1进程准备退出

str1进程调用exit（）函数进行退出，最终会映射到sys_exit（）函数去执行，并调用do_exi

（）函数来处理str1进程退出的相关事务。执行代码如下：

//代码路径: include/unistd.h:

volatile void exit (int status) ;

//代码路径: kernel/exit.c:

int sys_exit (int error_code)

{

return do_exit ((error_code&0xff) << 8) ;

}

进程退出包括两方面：第一，释放str1进程代码与数据所占用的物理内存并解除与str1这个可执行文件的关系，这一点由str1进程自己负责；第二，释放str1进程的管理结构task_struct所占用的物理内存并解除与task[64]的关系，这一点由父进程shell负责。

2.释放str1程序所占页面

进入do_exit () 函数后，调用free_page_tables () 函数将str1程序占用的页面释放掉，包括前面提到的已清栈但尚未释放的内存页面，并将管理这些页面的页表以及页目录项释放掉。这些页面中仍然保存着str1进程的垃圾数据，但解除了映射关系，str1进程将无法找到这些页面。执行代码如下：

```
//代码路径: kernel/exit.c:

int do_exit (long code)

{

    int i;

    free_page_tables (get_base (current->ldt[1]), get_limit
(0x0f)) ; //释放str1代码段所占用的页面

    free_page_tables (get_base (current->ldt[2]), get_limit
(0x17)) ; //释放str1数据段所占用的页面

    for (i=0; i<NR_TASKS; i++)
```

```

if (task[i] && task[i]->father==current->pid) {

task[i]->father=1;

if (task[i]->state==TASK_ZOMBIE)

/*assumption task[1]is always init*/

(void) send_sig (SIGCHLD,task[1], 1) ;

}

.....

}

```

释放过程如图6-20所示。

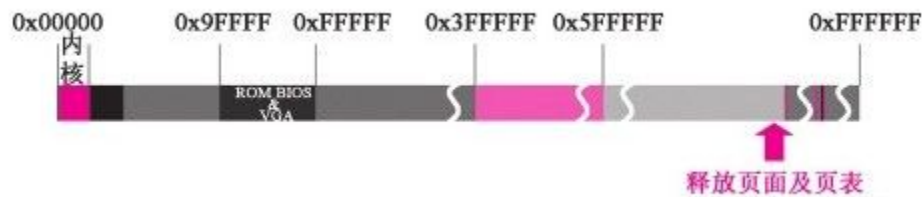


图 6-20 释放str1程序占用的页面

3.解除str1程序与文件有关的内容并给父进程发信号

解除该进程与str1程序对应的可执行文件的关系，具体表现为先将与父进程共享的文件释放掉，然后内核将str1进程设置为僵死状态，并给它的父进程shell发送“子进程退出”信号（信号处理的问题将在第8章中详细介绍）。对应代码如下：

```
//代码路径： kernel/exit.c:
```

```
int do_exit (long code)
```

```
{
```

```
.....
```

```
for (i=0; i<NR_OPEN; i++) //以下解除与父进程共享的文件
```

```
if (current->filp[i])
```

```
sys_close (i) ;
```

```
iput (current->pwd) ;
```

```
current->pwd=NULL;
```

```
iput (current->root) ;
```

```
current->root=NULL;

input (current->executable) ;

current->executable=NULL; //以上解除与父进程共享的文件

.....

current->state=TASK_ZOMBIE; //将str1设置为僵死状态

current->exit_code=code;

tell _father (current->father) ; //给父进程，即shell进程发信号

.....

}
```

关系解除并给父进程发信号的过程如图6-21所示。

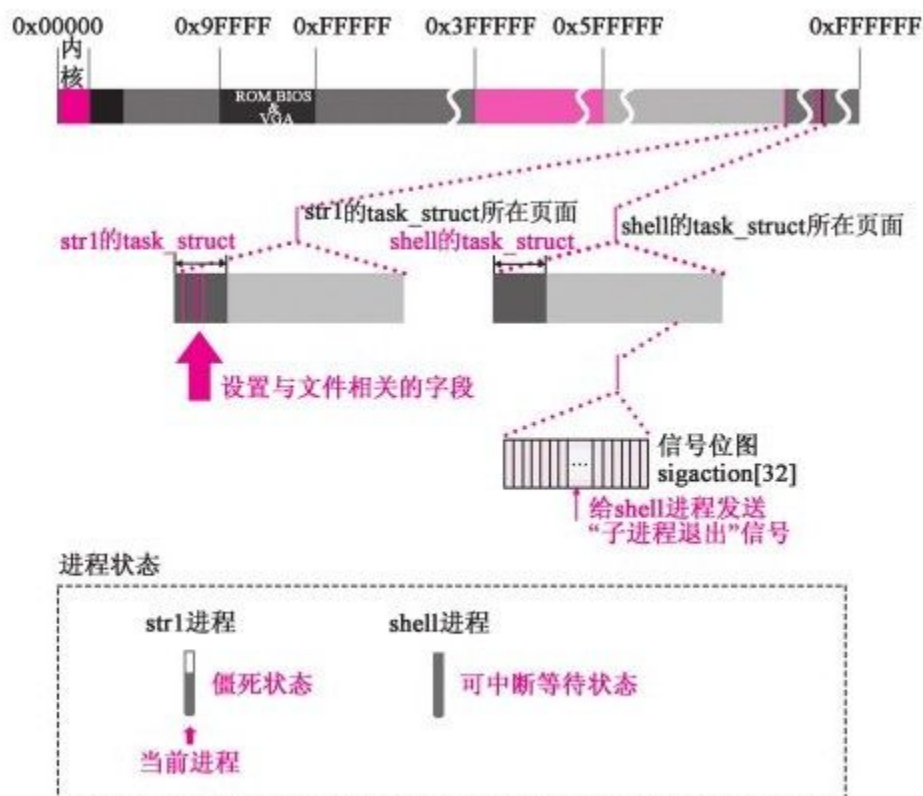


图 6-21 解除str1程序与文件有关的内容

4.str1程序退出后执行进程调度

到这里为止，str1进程对退出所做的善后工作已经完毕，str1进程将切换到其他进程去执行。由于现在只创建了一个用户进程，所以现在系统中

有进程0、进程1、updata进程、shell进程和str1这个用户进程。执行代码如下：

```
//代码路径: kernel/exit.c:

int do_exit (long code)

{

.....

current->state=TASK_ZOMBIE;

current->exit_code=code;

tell_father (current->father) ;

schedule () ; //准备切换到shell执行

return (-1) ; /*just to suppress warnings*/

}
```

进程切换效果如图6-22所示。



图 6-22 `str1`进程退出，切换到`shell`进程

`shell`进程接收到`str1`发送的信号而被唤醒，即设置为就绪态，之后切换到`shell`进程去执行。`shell`进程执行进入内核后，内核将释放掉`str1`进程 `task_struct`所占用的页面，并解除`str1`进程与 `task[64]`的关系，这样`str1`就彻底从系统中退出了。这个空出来的`task[64]`位置，可以被其他进程占用。占用了这个位置的进程，将具有和`str1`相同的线性地址空间和页目录项。

6.4 多个用户进程同时运行

本节以三个用户进程（str1、str2、str3）为例，来看看多个进程是如何运行的，它们又是如何切换的。

6.4.1 进程调度

1. 依次创建str1、str2和str3进程

我们假设系统中尚没有任何用户进程存在，外设上有三个可执行文件，分别是str1、str2、str3，且文件中的程序都与前面介绍的str1进程的代码一致。在此前提下，我们依次创建三个用户进程：str1、str2、str3。

现在last_pid已经累加为4了，所以这三个进程的进程号应该依次是5、6、7。现在task[64]中的前四项已经被占用了，所以这三个进程只能依次从第五项开始与task[64]建立关系，它们在线性地址空间中的位置应该依次是 $4 \times 64 \sim 5 \times 64$ MB、 $5 \times 64 \sim 6 \times 64$ MB、 $6 \times 64 \sim 7 \times 64$ MB。

这三个进程在线性空间中的分布效果如图6-23所示。

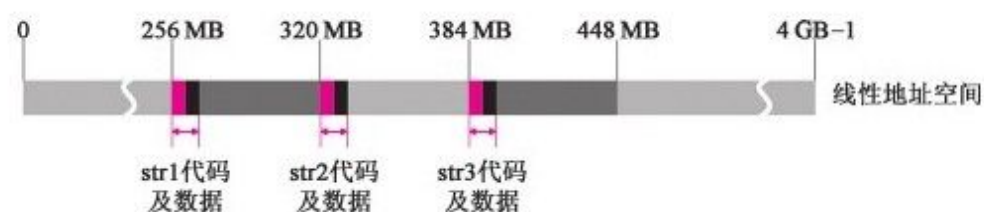


图 6-23 str1、str2和str3进程在线性空间中的分布

图6-24 展现了三个进程的task_struct和压栈的数据信息在物理内存中的分布情况。

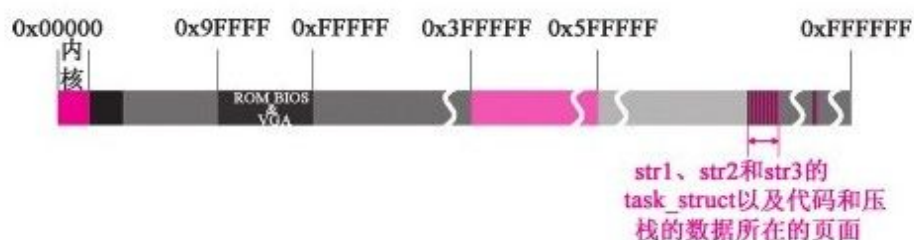


图 6-24 三个进程的task_struct和压栈数据信息在主内存区的分布

2.str1进程压栈的执行效果

假设现在轮到str1进程执行。str1开始执行foo函数调用，就需要压栈，于是产生缺页中断。在缺页中断处理中，内核为str1进程申请了空闲物理页面，并将其映射到str1进程的线性地址空间。之后，进程再对text数组进行设置，内容就被写在了刚分配的物理页面上了。

执行效果如图6-25所示。

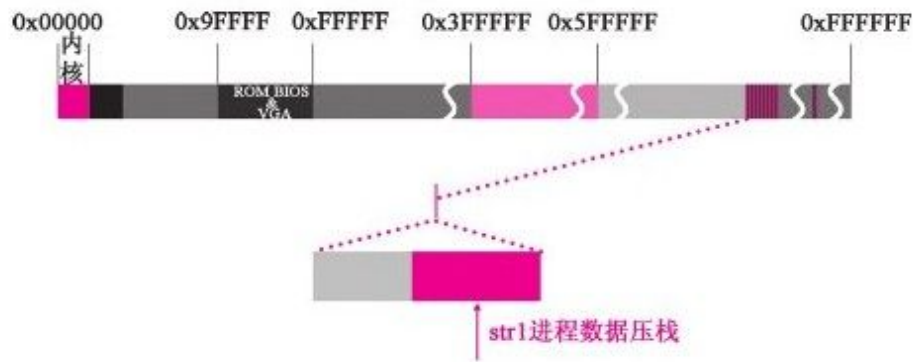


图 6-25 str1进程压栈的效果

3.str1运行过程中产生时钟中断并切换到str2执行

Linux 0.11中，两种情况下会导致进程切换。一种是由时钟中断引发的进程切换，这与进程执行毫无关系。无论是哪个进程执行，也无论是在3特权级下执行还是在0特权级下执行，时钟中断都会产生，只要满足切换条件，就切换。另一种是由进程执行引起的中断。进程执行到内核中时，

如果执行了类似读硬盘数据的程序，数据读出之前，进程无法继续执行，就应当将当前进程挂起，切换到其他进程去执行。但无论是哪种情况下的切换，都是TSS和LDT中的全套信息跟着进程走。下面先来看由时钟中断引发的切换。

str1在执行过程中，每10毫秒就会产生一次时钟中断，这样就会削减它的时间片。我们在程序中调用了sleep（）函数，以便起到延时的效果。这样当前进程的时间片被削减为0时，程序还没有执行完，此时要么是0特权级，要么是3特权级。str1进程一直在执行用户程序，特权级为3，此时就会调用schedule（）函数，准备进程切换。代码如下：

```
//代码路径：kernel/sched.c:
```



```
void do_timer (long cpl)
{
    .....

    if ( (--current->counter) > 0) return; //判断时间片是否削减为0

    current->counter=0;

    if (! cpl) return; //只有在3特权级下才能切换，0特权级下不能切
换

    schedule ();

}
```

于是切换至进程str2执行，进程str2也执行同样逻辑的程序。值得注意的是，当设置text数组时，屏幕打印的逻辑地址与当时进程str1的地址相同。但它们的线性地址不同，物理内存中进程str2也并没有与str1重叠。

str2执行压栈操作的效果如图6-26所示。

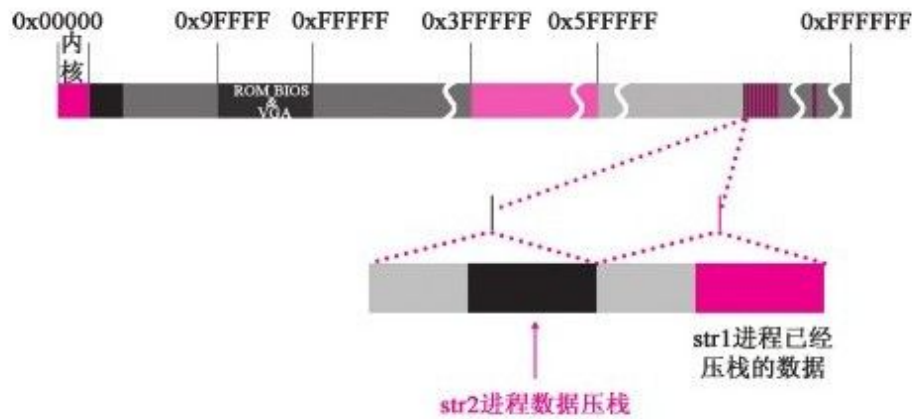


图 6-26 str2执行压栈操作的效果

4.str2执行过程中遇到时钟中断，切换到str3执行

str2执行一段时间后，时间片被削减为0后又切换到str3去执行。它也要压栈，str3开始运行，执行的代码与进程str2相同，也是压栈，并设置text。

str3程序执行压栈的效果如图6-27所示。

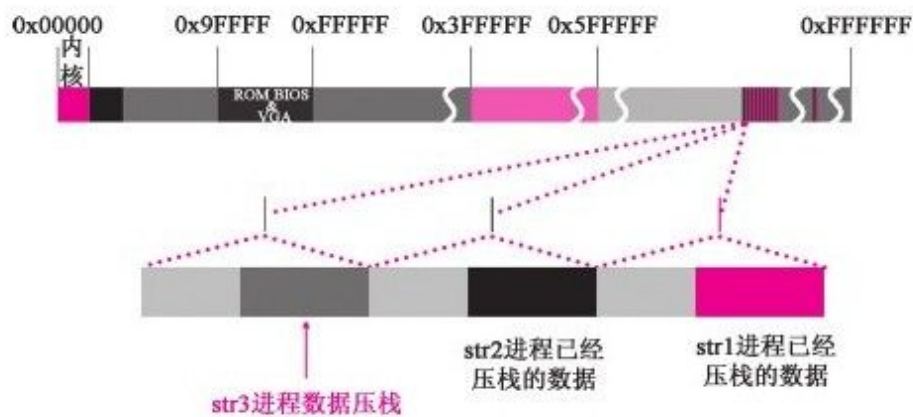


图 6-27 str3执行压栈操作的效果

我们不妨对str3的代码稍加改动，调用open（）、read（）和close（）函数，从硬盘上读文件。这样就映射到sys_read（）函数中执行。下达读盘指令后，数据不会马上进入缓冲区。没有数据，str3就不能继续执行。这时候它会主动地将自己挂起，然后切换到其他进程去执行。代码如下：

//代码路径： fs/buffer.c:

```
struct buffer_head * bread (int dev,int block)
```

```

{

struct buffer_head * bh;

if ( ! (bh=getblk (dev,block) ) )

panic ("bread: getblk returned NULL\n") ;

if (bh->b_uptodate)

return bh;

ll_rw_block (READ,bh) ;

wait_on_buffer (bh) ; //检查是否需要等待缓冲块解锁而将进程
挂起

if (bh->b_uptodate)

return bh;

brelse (bh) ;

return NULL;

}

static inline void wait_on_buffer (struct buffer_head * bh)

{

cli () ;

while (bh->b_lock)

```

sleep_on (&bh->b_wait) ; //缓冲块此时还在加锁，所以需要将进程挂起

sti () ;

}

void sleep_on (struct task_struct ** p)

{

.....

tmp=*p;

*p=current;

current->state=TASK_UNINTERRUPTIBLE; //将当前进程挂起，此时挂起的是str3

schedule () ; //切换到其他进程去执行

if (tmp)

tmp->state=0;

}

5.三个程序执行一段时间后在主内存的分布格局

`str3`执行一段时间后，时间片也用完了。这样三个用户进程虽然还需要继续执行，但时间片都用完了。当再发生时钟中断时，`do_timer()` 函数调用`schedule()` 函数进行进程切换，这时，内核会为它们重新分配时间片。

内核从`task[]`的末端开始重新给当前系统的所有进程（包括处于睡眠的进程，但进程0除外）分配时间片。时间片的大小为`counter/2+priority`。`priority`是进程的优先级，所以进程的优先级越高（`priority`值越大），分配到的时间片就越多。然后根据此时时间片的情况重新选择进程运行，如此反复进行。执行代码如下：

```
//代码路径: kernel/sched.c:
```

```
void schedule (void)
```

```
{  
  
.....  
  
for (p=&LAST_TASK; p > &FIRST_TASK; --p)  
  
if (*p)  
  
    (*p) -> counter= ( (*p) -> counter > 1) +  
  
    (*p) -> priority;  
  
.....  
  
}
```

这里值得注意的是，重新分配时间片时，并不需要给进程0进行分配。这是因为，只要系统中所有进程都暂时不具备执行条件，就自动切换到进程0去执行。进程0将一直执行下去，即便在此过程中它的时间片削减为0了，但由于还没有可以运行的进程，所以仍然需要进程0继续执行。这样一来，时间片对进程0来讲就没有意义了。可见，

进程0是一个特殊的进程，它的执行是由系统当前的留守需求决定的，时间片轮转这套机制并不适用于它。

接着它们都会继续不断地压栈，通过图6-28我们可以看出这一点。这三个用户进程在线性地址空间内压入它们各自的栈中的数据都是连续的，但在物理空间内压栈的数据却是完全“交错”分布的。

三个程序执行一段时间后，压入它们各自的栈中的数据在主内存区中的分布如图6-28所示。

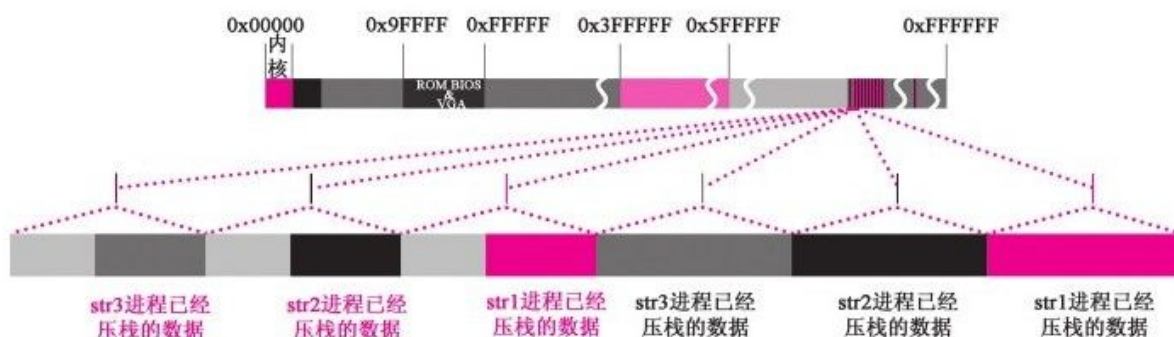


图 6-28 三个程序执行一段时间后压栈数据在主内存区的分布

不难发现，任何时刻都只有一个进程在执行，根本不存在多个进程同时执行的情况（所谓多进程同时执行，只是人的主观感受）。数据不会彼此覆盖，而且无论由于哪种情况产生进程切换，都通过调用`schedule()`函数来进行切换。这个函数进行进程切换，就会用TSS和LDT全套数据跟着进程走，以此实现对进程的保护。

6.4.2 页写保护

1. 进程A和进程B共享页面

假设现在系统有一个用户进程（进程A），它自己对应的程序代码已经载入内存中，此时该进程内存中所占用的页面引用计数都为“1”，接下来它开始执行，通过调用fork函数创建一个新进程（进程B）。在新进程创建的过程中，系统将进程A的页表项全部复制给进程B并设置进程B的页目录项。此时这两个进程就共享页面，被共享页面的引用计数累加为2，并将此共享页面全部设置为“只读”属性，即无论是进程A还是进程B，都只能对这些共享的页面进行读操作，而不能是写操作。执行代码如下：

//代码路径: mm/memory.c:

```
int copy_page_tables (unsigned long from,unsigned long to,long  
size)
```

```
{
```

```
.....
```

```
for (; nr-->0; from_page_table++, to_page_table++) {
```

```
    this_page=*from_page_table;
```

```
    if (! (1&this_page) )
```

```
        continue;
```

```
    this_page&=~2; //进程A对页面的操作属性被设置为只读
```

```
    *to_page_table=this_page; //进程A对页面的操作属性也被设置为  
只读
```

```
    if (this_page>LOW_MEM) {
```

```
        *from_page_table=this_page;
```

```
        this_page-=LOW_MEM;
```

```
        this_page>>=12;
```

```
        mem_map[this_page]++; //引用计数记录在mem_map中, 累加为2
```

```
}  
  
}  
  
.....  
  
}
```

页面共享的情况如图6-29所示。

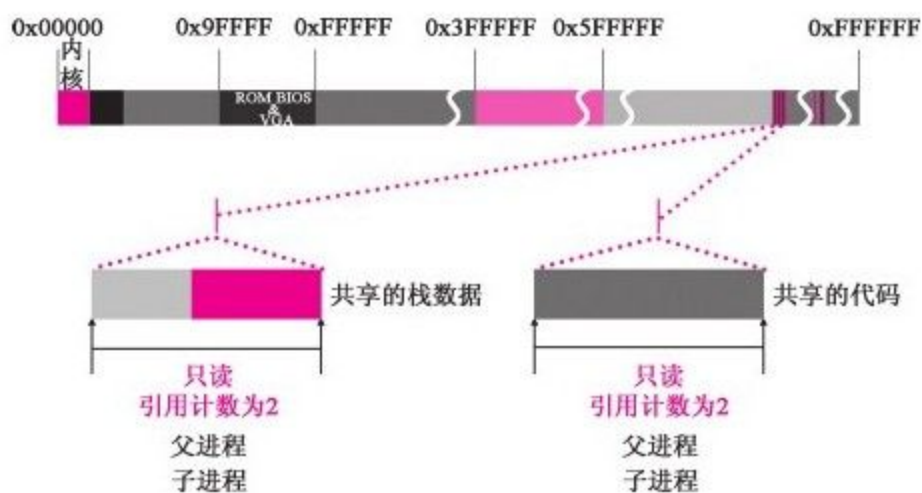


图 6-29 进程A与进程B共享页面的情况

2.进程A准备进行压栈操作

我们假设接下来轮到进程A执行，而且进程A接下来的动作是一个压栈动作，现在看看会发生什么。

现在进程A的程序对应的所有页面都是只读状态的。这就意味着，无论是代码所占用的页面，还是原先压栈的数据所对应的页面，都只能进行读操作，不能进行写操作。然而，压栈动作无疑是一个写操作，压栈时对应的线性地址值经过解析后肯定会映射到只读页面中，就会产生一个“页写保护”中断，如图6-30所示。

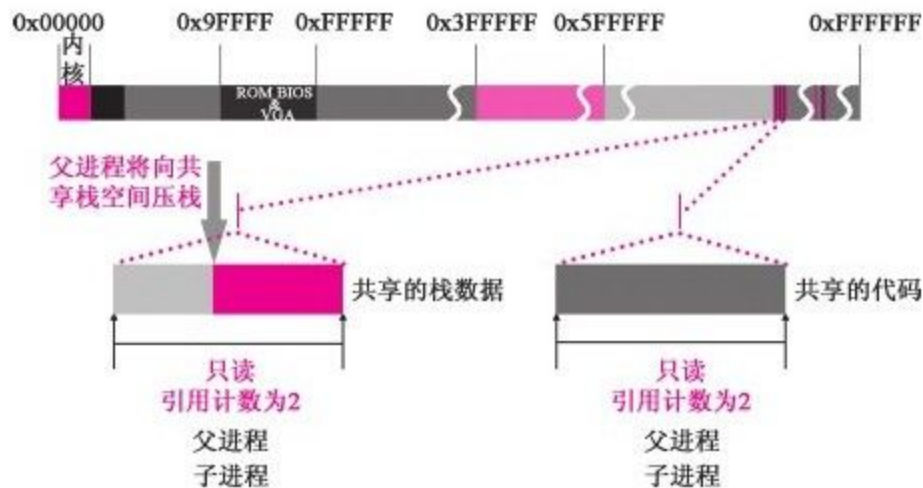


图 6-30 进程A准备执行压栈动作

3.进程A的压栈动作引发页面写保护

“页写保护”中断对应的服务程序是 `un_wp_page()` 函数，函数执行时，先要在主内存中申请一个空闲页面（以后我们称之为新页面），以便备份刚才压栈的位置所在页面（以后我们称之为原页面）的全部数据，然后将原页面的引用计数减1。这是因为，原页面中的数据即将要备份到新页面中了，进程A也将要到新页面中

操作数据，而不再需要与原页面维持关系了，所以原页面的引用计数减1。执行代码如下：

//代码路径：mm/memory.c:

```
void un_wp_page (unsigned long * table_entry)

{

.....

if ( ! (new_page=get_free_page ( ) ) ) //申请到新页面

oom ( ) ;

if (old_page >=LOW_MEM)

mem_map[MAP_NR (old_page) ]--; //页面引用计数递减1

.....

}
```

执行页写保护操作为进程A新申请一个页面之后的主内存分布如图6-31所示。

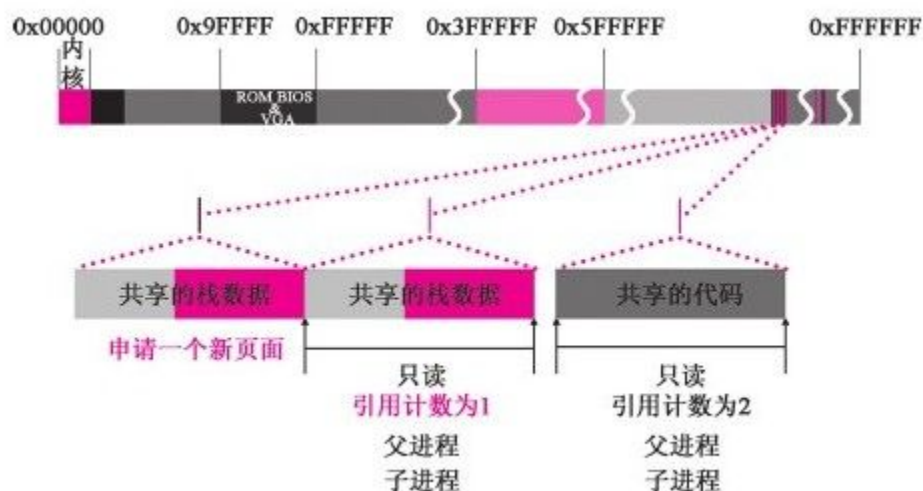


图 6-31 为进程A新申请一个页面存储压栈数据

值得注意的是，这里只是将原页面的引用计数减1，而并没有彻底释放。这是因为在整个操作系统中，所有可能被多个进程所共用的资源，比如文件i节点、文件管理表表项、内存页面等，都要通过引用计数来表示它们被使用的状况。当一个进程与它们解除关系后，其他进程未必都与它们解除了关系，简单的“释放”是不行的。

4.将进程A的页表指向新申请的页面

新页面虽然申请到了，但此时进程A的页表中与原页面对应的页表项还是指向原页面，没有页表项与页面对应，最终还是无法找到物理地址的。所以，现在还需要让进程A的页表中指向原页面的页表项改为指向新页面，并将其使用的属性从“只读”改变为“可读可写”，这样进程A才具备了在新页面中处理数据的能力。执行代码如下：

```
//代码路径: mm/memory.c:

void un_wp_page (unsigned long * table_entry)

{

.....

if (old_page >= LOW_MEM)

    mem_map[MAP_NR (old_page) ]--;

    *table_entry=new_page|7; //7的二进制形式为111，标志着新页面
    可读可写了

    invalidate ( ) ;
```

```
copy_page (old_page,new_page) ;  
  
}
```

此操作执行完毕后，进程A新分配的页面所处的状态如图6-32所示。

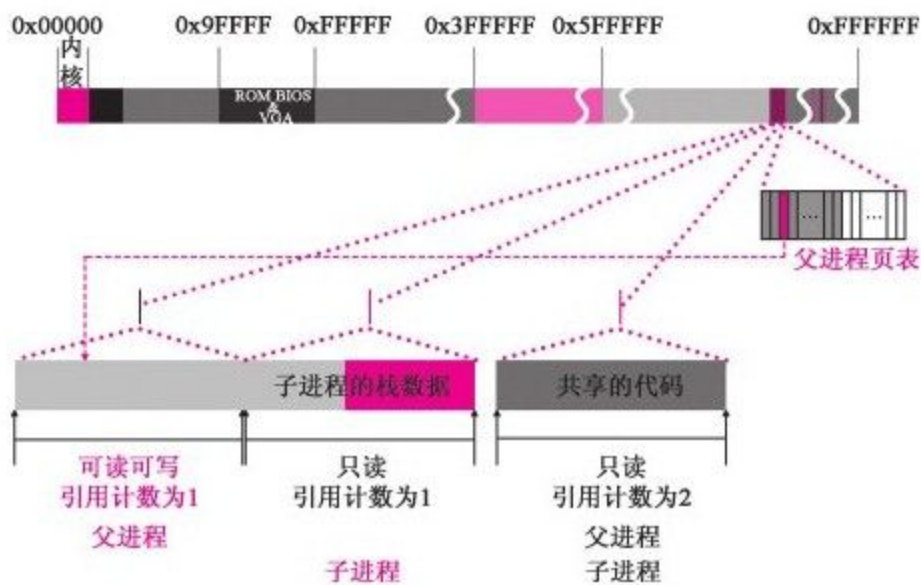


图 6-32 将进程A的页表与新申请页面对应

5.复制原页面内容到进程A新申请的页面

一切准备就绪后，就可以将原页面中的内容复制到新页面中了。复制之后，进程A就可以在新页面中完成这个压栈动作了。执行代码如下：

```
//代码路径: mm/memory.c:

void un_wp_page (unsigned long * table_entry)

{

.....

if (old_page >= LOW_MEM)

mem_map[MAP_NR (old_page) ]--;

*table_entry=new_page|7;

invalidate ( ) ;

    copy_page (old_page,new_page) ; //这里把原页面的数据复制到
新页面，供进程A使用

}
```

复制操作完成后，进程A新申请的内存页面中的存储情况如图6-33所示。

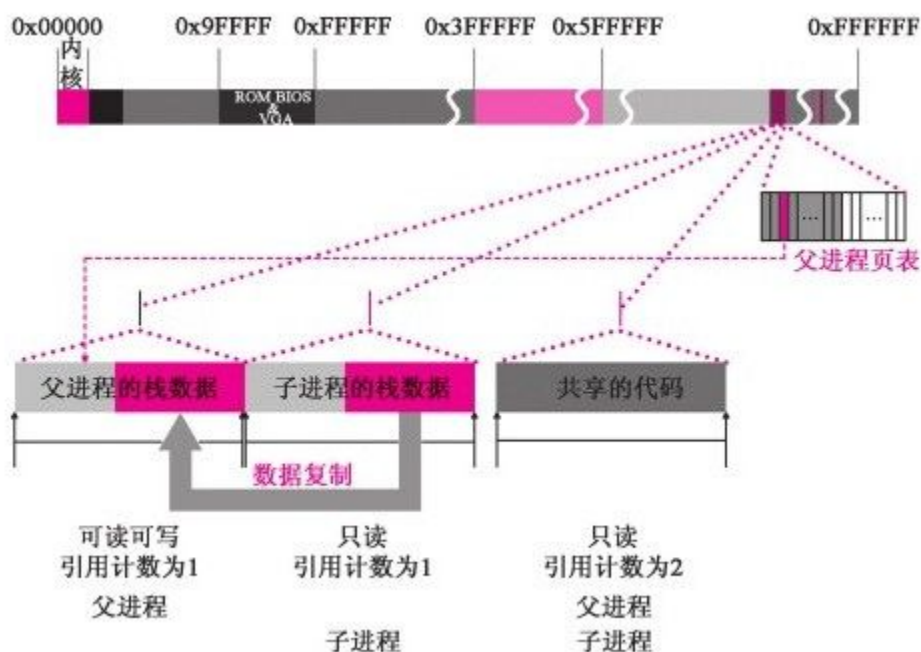


图 6-33 将原页面内容复制到进程A新申请的页面

6.进程B准备操作共享页面

进程A执行一段时间后，就该轮到它的子进程——进程B执行了。进程B仍然使用着原页面。

假设也要在原页面中进行写操作，但是现在原页面的属性仍然是“只读”的，这一点在进程A创建进程B时就是这样设置的，一直都没有改变过。所以在这种情况下，又需要进行页写保护处理，仍然是映射到`un_wp_page()`函数中。由于原页面的引用计数已经被削减为1了，所以现在就要将原页面的属性设置为“可读可写”，执行代码如下：

```
//代码路径: mm/memory.c:

void un_wp_page (unsigned long * table_entry)

{

.....

old_page=0xfffff000&*table_entry;

if (old_page>=LOW_MEM&&mem_map[MAP_NR
(old_page)]==1) {//发现原页面引用计数为1，不用共享了
```

写 *table_entry|=2; //2的二进制形式为010，R/W位设置为1，可读可

```
invalidate ();
```

```
return;
```

```
}
```

```
.....
```

```
}
```

进程A和进程B在压栈数据的处理方面可以各自操作不同的页面了。这些页面都是可读可写的，而且引用计数都为1，以后彼此都不会干扰对方，如图6-34所示。

现在进程B并没有自己的程序。如果将来它有了自己的程序，就会和原页面解除关系，原页面的引用计数将会继续减1，于是就变成0，系统将认定它为“空闲页面”。

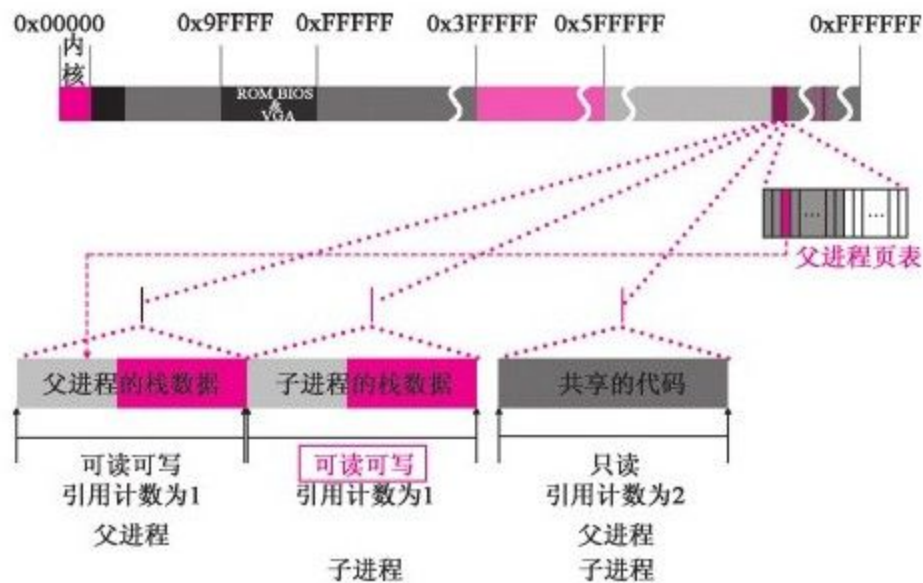


图 6-34 进程B将原共享页面性质修改

7.假设进程B先执行压栈操作的情况

我们重新假设，现在不是父进程——进程A先执行，而是子进程——进程B先执行，那么又会出现什么情况呢？

这种情况与前面所述的情况是对称的，即系统先为进程B申请一页面空间，然后让进程B的页表中与原页表相对应的页表项指向新页面，最后

将原页面内容复制给新页面，以便进程B操作。
等轮到进程A执行时，原页面被设置为“可读可写”，使进程A仍然使用原页面执行数据操作。

进程B先执行压栈操作时的内存分布如图6-35所示。

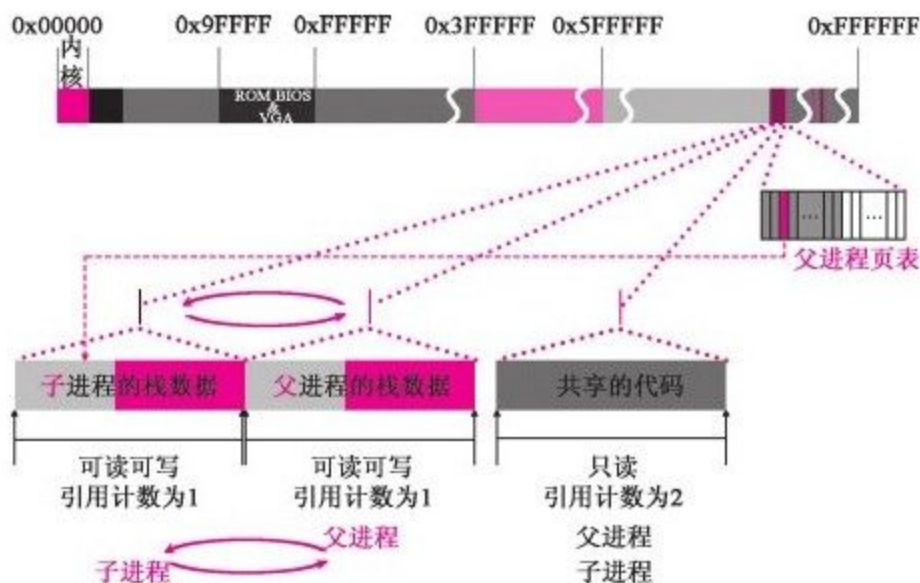


图 6-35 进程B先执行压栈操作时的内存分布

值得注意的是，页写保护是一个由内核执行的动作；在整个页写保护动作发生的过程中，用

户进程仍然正常执行，它并不知道自己在内存中被复制了，也不知道自己被复制到哪个页面了。

点评

从前面的讲解、分析，可以看出，在Linux 0.11（准确地说是UNIX体系）的设计者看来，所谓的操作系统就是若干正在操作的进程构成的系统，所谓内核只是进程的延续（所以才有用户态、内核态的说法），这是他们的设计指导思想。这个指导思想可以很好地解释大部分问题。但为进程划分线性地址空间、分页、时钟中断触发的进程调度……这些用“延续说”不太容易完全解释清楚。

我们将在本书的第9章尝试提出我们的设计指导思想。

6.5 本章小结

本章是全书中难度较大的一章，详细讲解了线性地址、分页、进程调度以及一个用户进程从创建到退出的完整过程和同时运行多个用户进程。

其中最难理解的是线性地址的部分。线性地址的设计思路体现在代码的字里行间，所以看似简单，实际很难掌握。理解线性地址需要整体思考操作系统各部分间的关系，要有想象力，可以说对线性地址理解的深度相当程度上影响对操作系统的理解水平。

与线性地址类似，分页机制的规则似乎也不复杂，但是涉及操作系统内核和用户进程运行的

很多方面，加之与线性地址的紧密关系，使得分页也是比较难以掌握的。

Linux操作系统设计者默认内核是进程的延续，所以深入理解、掌握用户进程运行的全过程及多进程同时运行时的进程调度，对理解、掌握Linux操作系统至关重要。

第7章 缓冲区和多进程操作文件

前面的章节已经讲解了进程、文件系统、内存管理。从这些讲解中我们能够感受到缓冲区横跨三者，作用非常重要。要想深刻理解操作系统，深刻理解进程、文件系统、内存管理之间的复杂关系，必须要搞清楚缓冲区的作用究竟是什么。

7.1 缓冲区的作用

要想清楚缓冲区的作用究竟是什么，不妨反过来思考：没有缓冲区行不行？没有缓冲区会遇到什么麻烦？

从计算机的物理层面上看，缓冲区是在物理内存中开辟的一块空间，这块内存空间的物理性质与进程所占的内存空间没有什么本质的不同。块设备（为了方便，本章只讨论硬盘）与缓冲区交互数据同硬盘与进程内存空间交互数据从物理层面上看完全相同，既不会影响交互数据的正确性，也不会影响交互数据的传输速度。从这个角度看，没有缓冲区没有什么不可以，完全可以实现进程与硬盘的数据交互。

由此可见，缓冲区不是必须的，设计缓冲区一定是为了使操作系统运行地更好（锦上添花）。

设计缓冲区究竟能给操作系统的运行带来哪些好处？

我们认为主要体现在两方面：

1) 形成所有块设备数据的统一集散地，操作系统的设计更方便、更灵活。

2) 对块设备的文件操作运行效率更高。

第1方面比较容易理解，第2方面是理解操作系统的难点之一。所以，这一章我们将通过两个多进程操作文件实例，详细讲解缓冲区在提高块设备操作文件运行效率方面的设计。

仔细观察图7-1，可以发现这里面似乎有一个问题，进程内存空间与缓冲区内存空间是同样的内存。进程内存空间与硬盘交换数据的时候，中间加上一个缓冲区，只会增加一次数据在内存中倒手的时间，而这次数据倒手没有任何数据处

理，只是简单的倒手，应该只会增加对CPU资源的消耗，怎么会比进程直接与硬盘交换数据还快呢？

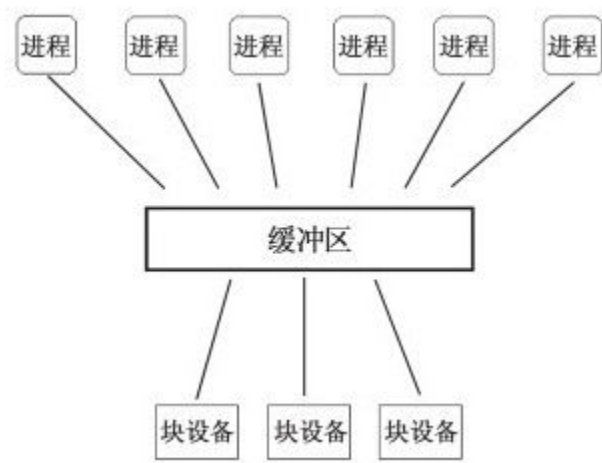


图 7-1 进程、缓冲区、块设备的格局图

快的原因是缓冲区的共享。在计算机中，内存与内存的数据交互的速度是内存与硬盘数据交互速度的2个量级。如果A进程从硬盘读到缓冲区的数据，恰好也是B进程需要读取的，B进程就不用从硬盘读取，可以直接从缓冲区中读取，B进

程所花费的时间大约只有A进程读取这个数据的百分之一。效率一下子提高了2个量级。如果还有C进程、D进程、E进程.....恰好都需要读取这个数据，计算机的整体效率就会大大提高。这是缓冲区共享的一种模式，就是不同进程之间共享缓冲区中的数据。如果A进程读取、使用完这个数据，过了一段时间需要再一次读取这个数据，此时这个数据还停留在缓冲区，A进程就可以直接从缓冲区中读取，不需要花费100倍的时间从硬盘上读取。这是共享的另一种模式，就是同一个进程在不同时间多次共享缓冲区中的同一个数据。再有就是这两种模式的组合模式。以上分析的是读操作的共享。写操作的共享与此类似。

仔细思考上面的分析，可以发现，要想通过缓冲区的设计提高操作系统读写文件的整体效率，就应该尽可能多地共享缓冲区中的数据。而尽可能多地共享缓冲区中的数据，最有效、最直接的方法就是让缓冲区中的数据在缓冲区中停留的时间尽可能长！

可以说，缓冲区的所有的代码都是围绕着如何保证数据交互的正确性、如何让数据在缓冲区中停留的时间尽可能长来设计的。本章后续的内容将通过两个实例，详细讲解操作系统代码是如何实现“让数据在缓冲区中停留的时间尽可能长”这个目标的。

7.2 缓冲区的总体结构

缓冲区涉及进程、内存、文件，内容很多，代码庞杂，很不容易看懂，是理解操作系统的难点之一。为了更好地学习、掌握缓冲区的设计，我们先俯瞰缓冲区的总体结构，如图7-2所示。

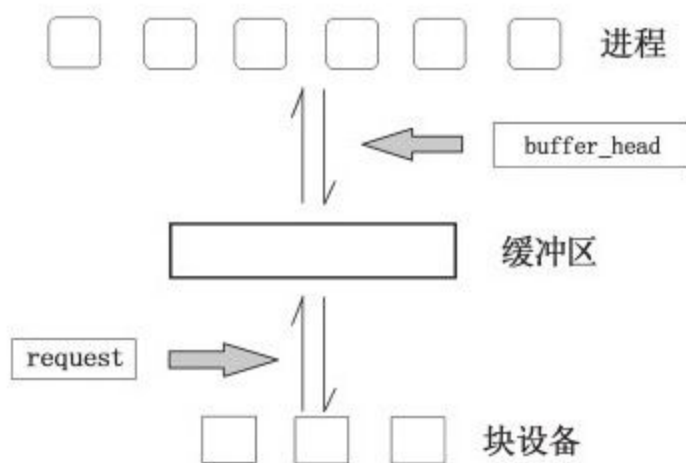


图 7-2 缓冲区、buffer_head、request的格局图

在Linux操作系统中，为缓冲区配套地设计了两个非常重要的管理信息： `buffer_head`、`request`。其中`buffer_head`主要负责进程与缓冲区中的缓冲块的数据交互，在确保数据交互正确的前提下，让数据在缓冲区中停留的时间尽可能长；`request`主要负责缓冲区中的数据与块设备之间的数据交互，在确保数据交互正确的前提下，尽可能及时地将进程修改过的缓冲块中的数据同步到块设备上。

这两个管理信息的数据结构代码如下：

```
//代码路径：include/linux/fs.h:

struct buffer_head{

char * b_data; /*pointer to data block (1024 bytes) */

unsigned long b_blocknr; /*block number*/
```

```
unsigned short b_dev; /*device (0=free) */

unsigned char b_uptodate;

unsigned char b_dirt; /*0-clean, 1-dirty*/

unsigned char b_count; /*users using this block*/

unsigned char b_lock; /*0-ok, 1-locked*/

struct task_struct * b_wait;

struct buffer_head * b_prev;

struct buffer_head * b_next;

struct buffer_head * b_prev_free;

struct buffer_head * b_next_free;

};
```

//代码路径: kernel/blk_drv/blk.h:

```
struct request{

int dev; /*-1 if no request*/

int cmd; /*READ or WRITE*/

int errors;

unsigned long sector;
```

```
unsigned long nr_sectors;  
  
char * buffer;  
  
struct task_struct * waiting;  
  
struct buffer_head * bh;  
  
struct request * next;  
  
};
```

下面将详细讲解为什么要设计这个数据结构，以及这个数据结构是如何做到“让数据在缓冲区中停留的时间尽可能长”的。

7.3 b_dev、b_blocknr及request的作用

b_dev和b_blocknr这两个字段是buffer_head结构中非常重要的两个字段，是缓冲区支持多进程共享文件的基础。它们既是正确性的基础，又是“让数据在缓冲区中停留的时间尽可能长”的基础。下面先来介绍这两个字段是如何确保正确性的。

7.3.1 保证进程与缓冲块数据交互的正确性

进程与缓冲区不是以文件为单位、而是以缓冲块为单位进行数据交互的，一次交互若干个块，数据不足一个缓冲块的仍占用一个缓冲块。

缓冲区与硬盘的交互也是以缓冲块为单位的，而且缓冲块与硬盘块的单位大小一致。进程操作文件时，由进程提出的文件操作请求，最终由操作系统落实到与硬盘上具体某个数据块进行交互。有了缓冲区，进程和硬盘上的数据块不是直接交互数据，而是通过缓冲块再进行交互。保证数据交互的正确性，首先要保证硬盘上的数据块与缓冲块必须严格对应。

因为硬盘的设备号、块号能唯一确定硬盘块，第2章讲解过每一个缓冲块都有唯一的一个 `buffer_head` 管理，所以，操作系统采取的策略是，内核通过 `buffer_head` 结构中的 `b_dev` 和 `b_blocknr` 两个字段，把缓冲块和硬盘数据块的关系绑定。这样，就保证了硬盘块与缓冲块关系的

唯一性，进程、缓冲块的数据交互与进程、硬盘数据块的数据交互等价，确保数据交互不会出现混乱。代码如下：

```
//代码路径: fs/buffer.c:

struct buffer_head * getblk (int dev,int block) //申请缓冲块

{

repeat:

    if (bh=get_hash_table (dev,block) ) //如果发现缓冲块与指定设备 (dev) 指定数据块 (block) 已经绑定

return bh; //返回，直接使用

    tmp=free_list; //如果没找到符合指定标准的绑定缓冲块，就申请新缓冲块

do{

    if (tmp->b_count)

continue;

    if (! bh||BADNESS (tmp) < BADNESS (bh) ) {

bh=tmp;
```

```

if ( ! BADNESS (tmp) )

break;

/*and repeat until we find something good*/

}while ( (tmp=tmp->b_next_free) !=free_list) ;

.....

/*OK,FINALLY we know that this buffer is the only one of it's kind,
*/

/*and that it's unused (b_count=0) , unlocked (b_lock=0) , and
clean*/

bh->b_count=1;

bh->b_dirt=0;

bh->b_uptodate=0;

remove_from_queues (bh) ;

bh->b_dev=dev; //对新缓冲块的设备号进行设置

bh->b_blocknr=block; //对新缓冲块的块号进行设置

insert_into_queues (bh) ;

return bh;

}

```

从以上代码中可以看出，新申请缓冲块的时候就将缓冲块与数据块的关系锁死，使得内核在进程方向上，只要将文件的操作位置确定，并将其转换成b_dev、b_blocknr就可以了，硬盘数据块与缓冲块的关系不用考虑，最终与硬盘的交互肯定是正确的。

读文件时，内核通过文件操作指针计算出文件数据内容所在的b_dev、b_blocknr，进程方面的延伸到缓冲块为止，执行bread（）函数以后就不再与硬盘数据块直接打交道了。代码如下：

//代码路径：fs/file_dev.c:

```
int file_read (struct m_inode * inode,struct file * filp,char * buf,int
count) //读文件
{
```

```

.....

if ( (left=count) <=0)

return 0;

while (left) {

    if (nr=bmap (inode, (filp->f_pos) /BLOCK_SIZE) ) {//通过
文件偏移指针, 计算出块号

        if (! (bh=bread (inode->i_dev,nr) ) ) //实参中inode->i_dev就
是设备号, nr就是块号

            break;

        }else

            bh=NULL;

        nr=filp->f_pos%BLOCK_SIZE;

        chars=MIN (BLOCK_SIZE-nr,left) ;

        .....

    }

    .....

}

//代码路径: fs/buffer.c:

```

```

struct buffer_head * bread (int dev,int block) //读取底层块设备数据
{
    struct buffer_head * bh;

    if (! (bh=getblk (dev,block) ) ) //申请缓冲块时要用到设备号
和块号

        panic ("bread: getblk returned NULL\n") ;

    if (bh->b_uptodate)

        return bh;

    .....

}

```

与读文件类似，写文件时，内核通过文件操作指针计算出文件数据内容所在的**b_dev**和**b_blocknr**，进程方面的延伸到此为止。代码如下：

//代码路径： fs/file_dev.c:

```

int file_write (struct m_inode * inode,struct file * filp,char * buf,int
count) //写文件{

.....

if (filp->f_flags&O_APPEND)

pos=inode->i_size;

else

pos=filp->f_pos;

while (i<count) {

    if (! (block=create_block (inode,pos/BLOCK_SIZE) ) ) //通过
文件偏移指针，计算出块号

        break;

    if (! (bh=bread (inode->i_dev,block) ) ) //实参中inode->
i_dev就是设备号，nr就是块号

        break;

    c=pos%BLOCK_SIZE;

    p=c+bh->b_data;

    bh->b_dirt=1;

    .....

}

```

```

.....

}

//代码路径: fs/buffer.c:

struct buffer_head * bread (int dev,int block) //读取底层块设备数据

{

    struct buffer_head * bh;

    if ( ! (bh=getblk (dev,block) ) ) //申请缓冲块时要用到设备号
和块号

        panic ("bread: getblk returned NULL\n" ) ;

    if (bh->b_uptodate)

        return bh;

    .....

}

```

进程方向上交互文件内容时如此，交互文件管理信息时也是如此。

内核读取i节点时，通过i节点号以及超级块中的信息，计算出i节点所在的b_dev和b_blocknr，而不会跨过缓冲块直接操作硬盘数据块。代码如下：

```
//代码路径: fs/inode.c:

static void read_inode (struct m_inode * inode) //读i节点
{
    .....

    lock _inode (inode) ;

    if ( ! (sb=get_super (inode->i_dev) ) )

        panic ("trying to read inode without dev") ;

    block=2+sb->s_imap_blocks+sb->s_zmap_blocks+//通过i节点号和
    超级块信息，确定了block (块号)

    (inode->i_num-1) /INODES_PER_BLOCK;

    if ( ! (bh=bread (inode->i_dev,block) ) ) //实参中inode->
    i_dev就是设备号， nr就是块号
```



```

panic ("unable to read i-node block") ;

* (struct d_inode *) inode=

( (struct d_inode *) bh->b_data)

[ (inode->i_num-1) %INODES_PER_BLOCK]; //从缓冲块中提
取i节点，载入inode_table[32]

brelse (bh) ;

unlock_inode (inode) ;

}

//代码路径: fs/buffer.c:

struct buffer_head * bread (int dev,int block) //读取底层块设备数据

{

struct buffer_head * bh;

if (! (bh=getblk (dev,block) ) ) //申请缓冲块时要用到设备号
和块号

panic ("bread: getblk returned NULL\n") ;

if (bh->b_uptodate)

return bh;

.....

```

```
}
```

与内核读i节点类似，内核写入i节点时，通过i节点号以及超级块中的信息，计算出i节点所在的b_dev和b_blocknr，动作到此为止。代码如下：

```
//代码路径： fs/inode.c:
```

```
static void write_inode (struct m_inode * inode) //写i节点
```

```
{
```

```
.....
```

```
if (! (sb=get_super (inode->i_dev) ) )
```

```
panic ("trying to write inode without device") ;
```

```
    block=2+sb->s_imap_blocks+sb->s_zmap_blocks+//通过i节点号和  
    超级块信息，确定了block（块号）
```

```
    (inode->i_num-1) /INODES_PER_BLOCK;
```

```
    if (! (bh=bread (inode->i_dev,block) ) ) //实参中inode->  
    i_dev就是设备号， nr就是块号
```

```
panic ("unable to read i-node block") ;
```

```

    ( (struct d_inode *) bh->b_data)

[ (inode->i_num-1) %INODES_PER_BLOCK]=

* (struct d_inode *) inode; //从inode_table[32]中提取i节点，载入
缓冲块

bh->b_dirt=1;

inode->i_dirt=0;

.....

}

//代码路径: fs/buffer.c:

struct buffer_head * bread (int dev,int block) //读取底层块设备数据

{

    struct buffer_head * bh;

    if ( ! (bh=getblk (dev,block) ) ) //申请缓冲块时要用到设备号
和块号

        panic ("bread: getblk returned NULL\n" ) ;

    if (bh->b_uptodate)

        return bh;

    .....

```

```
}
```

同样，内核加载超级块时，通过传递下来的设备号和指定的块号，计算出超级块、i节点位图、逻辑块位图所在的b_dev和b_blocknr，动作延伸至此。代码如下：

```
//代码路径: fs/super.c:

static struct super_block * read_super (int dev) //读取超级块

{

.....

s->s_time=0;

s->s_rd_only=0;

s->s_dirt=0;

lock_super (s) ;

if ( ! (bh=bread (dev, 1) ) ) { //1就是块号，超级块是设备中1
号数据块
```

```

s->s_dev=0;

free_super (s) ;

return NULL;

}

* ( (struct d_super_block *) s) =

* ( (struct d_super_block *) bh->b_data) ;

brelse (bh) ;

if (s->s_magic != SUPER_MAGIC) {

s->s_dev=0;

free_super (s) ;

return NULL;

}

.....

block=2; //2是第一个i节点位图的块号

for (i=0; i<s->s_imap_blocks; i++)

if (s->s_imap[i]=bread (dev,block) )

block++;

```

```
else
```

```
break;
```

```
for (i=0; i<s->s_zmap_blocks; i++) //block不断累加，依据此  
加载超级块位图
```

```
if (s->s_zmap[i]=bread (dev,block) )
```

```
block++;
```

```
else
```

```
break;
```

```
if (block!=2+s->s_imap_blocks+s->s_zmap_blocks) {
```

```
for (i=0; i<I_MAP_SLOTS; i++)
```

```
breakelse (s->s_imap[i]) ;
```

```
.....
```

```
}
```

```
//代码路径: fs/buffer.c:
```

```
struct buffer_head * bread (int dev,int block) //读取底层块设备数据
```

```
{
```

```
struct buffer_head * bh;
```

```
    if ( ! (bh=getblk (dev,block) ) ) //申请缓冲块时要用到设备号
和块号

    panic ("bread: getblk returned NULL\n") ;

    if (bh->b_uptodate)

    return bh;

    .....

}
```

以上代码表明，进程方向上，只要把缓冲块对应的设备号和块号确定，正确性就有保证，`getblk ()` 函数包揽了绑定的工作。

从硬盘方向看，内核通过另一个数据结构（`request`）来支持缓冲块与硬盘的交互，请求项中设备号`dev`和块的首扇区号`sector`（块是操作系统的概念，硬盘只有扇区的概念）决定了数据交互的位置。而这两个字段的数值，也是通过

buffer_head中b_dev和b_blocknr两个字段的值来设置的。这说明，只要缓冲块的设备号和块号确定了，内核通过请求项与硬盘交互时，最多考虑到缓冲区就足够了，不需要再往进程方面延伸，去考虑进程的文件操作问题。代码如下：

```
//代码路径: kernel/blk_drv/ll_rw_blk.c:
```

```
void ll_rw_block (int rw,struct buffer_head * bh) //底层块设备操作
{
    unsigned int major;

    if ( (major=MAJOR (bh->b_dev) ) >=NR_BLK_DEV||

        ! (blk_dev[major].request_fn) ) {

        printk ("Trying to read nonexistent block-device\n\r") ;

        return;

    }

    make_request (major,rw,bh) ; //设置请求项
```



```

}

static void make_request (int major,int rw,struct buffer_head * bh)

{

.....

if (req<request) {

if (rw_ahead) {

unlock_buffer (bh) ;

return;

}

sleep_on (&wait_for_request) ;

goto repeat;

}

req->dev=bh->b_dev; //用缓冲块中的b_dev设置请求项

req->cmd=rw;

req->errors=0;

req->sector=bh->b_blocknr<<1; //用缓冲块中的b_blocknr设置
请求项

```

```
req->nr_sectors=2;
```

```
req->buffer=bh->b_data;
```

```
req->waiting=NULL;
```

```
req->bh=bh;
```

```
req->next=NULL;
```

```
add_request (major+blk_dev,req) ; //加载请求项
```

```
}
```

```
static void add_request (struct blk_dev_struct * dev,struct request *  
req)
```

```
{
```

```
.....
```

```
if ( ! (tmp=dev->current_request) ) {
```

```
dev->current_request=req;
```

```
sti () ;
```

```
(dev->request_fn) () ; //下达硬盘操作命令
```

```
return;
```

```
}
```

```
.....
```

```
}
```

```
//代码路径: kernel/blk_drv/hd.c:
```

```
void do_hd_request (void)
```

```
{
```

```
.....
```

```
INIT_REQUEST;
```

```
dev=MINOR (CURRENT->dev) ; //从请求项中获取设备号
```

```
block=CURRENT->sector; //从请求项中获取块号
```

```
if (dev >= 5*NR_HD || block+2 > hd[dev].nr_sects) {
```

```
end_request (0) ;
```

```
goto repeat;
```

```
}
```

```
.....
```

```
__asm__
```

```
("divl%4": "=a" (block) , "=d" (sec) : "0" (block) , "1" (0) , /  
/通过块号//来换算磁头、扇区、柱面等参数
```

```
"r" (hd_info[dev].sect) ) ;
```

```

__asm__
("divl%4": "=a" (cyl) , "=d" (head) : "0" (block) , "1" (0) ,

    "r" (hd_info[dev].head) ) ;

sec++;

nsect=CURRENT->nr_sectors;

.....

}

```

综上所述，在进程方向上，任何复杂的文件操作，比如对文件任意部分修改、插入或删除数据等，只要通过这两个字段和数据块锁死，就能保证正确性；从进程方向看，与缓冲块交互等价于与硬盘数据块交互。

图7-3反映了加写的数据正好在一个块内的情景。



图 7-3 块内加写情景

图7-4反映了块间加写数据的情景。

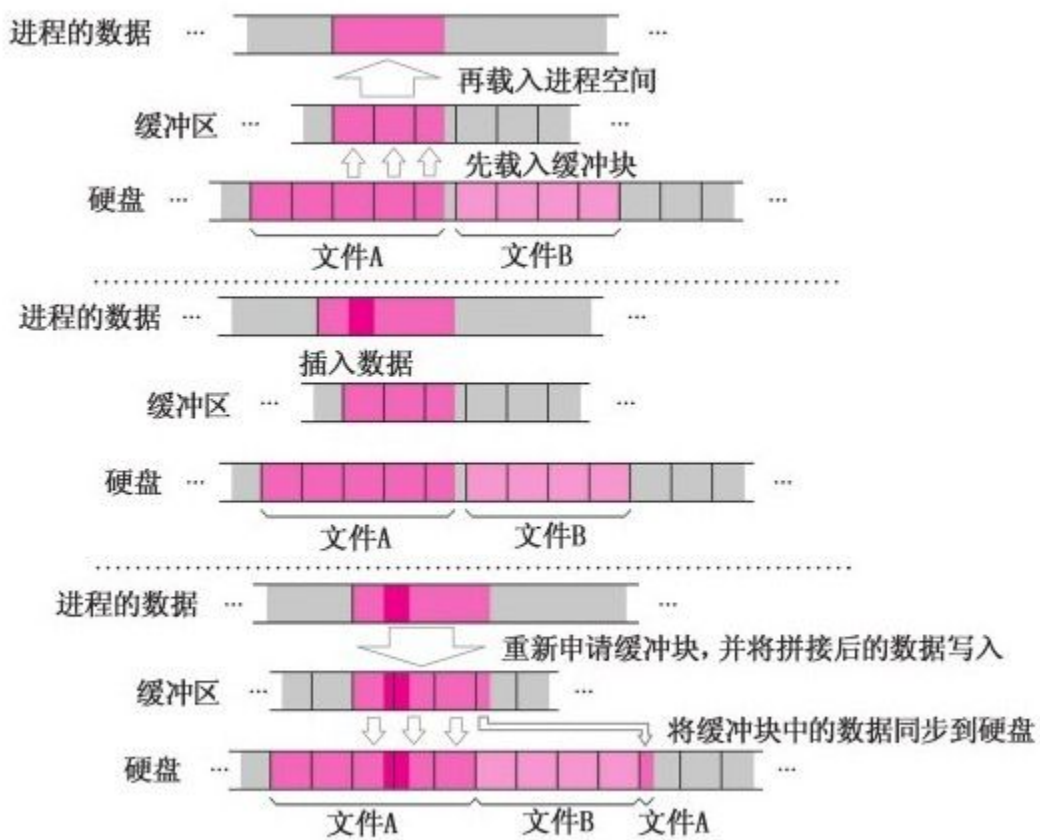


图 7-4 块间加写情景

7.3.2 让数据在缓冲区中停留的时间尽可能长

`b_dev`和`b_blocknr`不仅保证了正确性，而且是数据在缓冲区中多停留一段时间的基础。

数据是否停留在缓冲区中的标志是，缓冲块是否还与硬盘的数据块存在绑定关系。代码如下：

```
//代码路径： fs/buffer.c:

struct buffer_head * getblk (int dev,int block) //申请缓冲块

{

    struct buffer_head * tmp, *bh;

    repeat:

        if (bh=get_hash_table (dev,block) ) //试图从尚且存在绑定关系的
缓冲块中沿用
```

```

return bh;

tmp=free_list;

.....

}

struct buffer_head * get_hash_table (int dev,int block)

{

.....

for ( ; ) {

    if ( ! (bh=find_buffer (dev,block) ) ) //查找设备号和块号符合
要求的缓冲块

return NULL;

bh->b_count++;

wait_on_buffer (bh) ;

.....

}

}

static struct buffer_head * find_buffer (int dev,int block)

{

```



```
struct buffer_head * tmp;

for (tmp=hash (dev,block) ; tmp !=NULL; tmp=tmp->
b_next) //遍历哈希表进行比对

if (tmp->b_dev==dev&&tmp->b_blocknr==block)

return tmp; //如果找到现成的，就返回tmp

return NULL; //如果没找到现成的，就返回NULL

}
```

从上面代码中可以看出，内核从哈希表中搜索现成的缓冲块时，只看设备号和块号，其他的什么都不管。只要缓冲块与硬盘数据块的绑定关系还存在，就认定数据块中的数据仍然停留在缓冲块中，就可以直接用，不需要从硬盘上读取，节省了100倍的硬盘读取时间。

如果在缓冲区中找了一遍，所有缓冲块都已经与硬盘中的数据块建立了绑定关系，但b_dev、

b_blocknr又都不是进程所需要的，实在找不到合适的，那就只能找一个暂时没有进程使用

(b_count为0) 的空闲缓冲块，废掉已经存在的绑定关系，用新的绑定关系取而代之，即新建缓冲块。到此为止，硬盘数据块中的数据才算不在缓冲块中停留了。代码如下：

//代码路径： fs/buffer.c:

```
struct buffer_head * getblk (int dev,int block) //申请缓冲块
```

```
{
```

```
.....
```

```
if (find_buffer (dev,block) )
```

```
goto repeat;
```

```
bh->b_count=1;
```

```
bh->b_dirt=0;
```

```
bh->b_uptodate=0;
```

```
remove_from_queues (bh) ;  
  
bh->b_dev=dev;  
  
bh->b_blocknr=block;  
  
insert_into_queues (bh) ;  
  
return bh;  
  
}
```

这两行代码的意思是，新申请的缓冲块与数据块建立绑定关系。新申请一个缓冲块会有两种情景。一种是操作系统刚启动的时候，该缓冲块没有和任何数据块建立绑定关系。另一种是操作系统已经运行了一段时间，执行过充分多的文件读写操作，所有缓冲块都与硬盘数据块建立了绑定关系，而且所有的缓冲块与新申请的缓冲块的b_dev、b_blocknr不符，不能共享。所以只能在暂时没有进程使用（b_count为0）的缓冲块中，强行

占用一个缓冲块。在这种情景下，这两行代码就有两层意思：

- 1) 建立一个新的缓冲块与硬盘数据块之间的绑定关系产生。

- 2) 废除之前的缓冲块与硬盘数据块的绑定关系。

值得注意的是，为了让数据块中的数据尽可能长时间地停留在缓冲块中，内核中没有任何机制、也没有任何代码能够将已经建立了绑定关系的缓冲块、硬盘数据块，刻意地、主动地解除绑定关系。只有在迫不得已的情况下，才被新建立的绑定关系强行替换。所有这些的目的只有一个，就是尽可能使数据在缓冲区中停留更长的时间。现在，我们可以看出，`b_dev`和`b_blocknr`是硬

盘数据块的时间在缓冲区中停留更长时间的非常重要的管理信息。

反观请求项request的设计思路，和缓冲区正好相反，它的目的是，尽可能快地让缓冲块和硬盘交互数据。前面我们介绍到，request中也有与b_dev和b_blocknr类似的字段，它们就是设备号dev和块的首扇区号sector。它们除了保证缓冲块和硬盘数据块交互的正确性外，更多的是尽可能快地让缓冲块和数据块进行交互。我们来看如下代码：

//代码路径：kernel/blk_drv/ll_rw_blk.c:

```
void ll_rw_block (int rw,struct buffer_head * bh) //底层块设备操作
{
    unsigned int major;
```

```

if ( (major=MAJOR (bh->b_dev) ) >=NR_BLK_DEV||

! (blk_dev[major].request_fn) ) {

printk ("Trying to read nonexistent block-device\n\r") ;

return;

}

make_request (major,rw,bh) ; //设置请求项

}

static void make_request (int major,int rw,struct buffer_head * bh)

{

.....

if (req<request) {

if (rw_ahead) {

unlock_buffer (bh) ;

return;

}

sleep_on (&wait_for_request) ;

goto repeat;

```

```

    }

    req->dev=bh->b_dev; //用缓冲块中的b_dev设置请求项

    req->cmd=rw;

    req->errors=0;

    req->sector=bh->b_blocknr<<1; //用缓冲块中的b_blocknr设置
请求项

    req->nr_sectors=2;

    req->buffer=bh->b_data;

    req->waiting=NULL;

    req->bh=bh;

    req->next=NULL;

    add_request (major+blk_dev,req) ; //加载请求项

}

static void add_request (struct blk_dev_struct * dev,struct request *
req)

{

    struct request * tmp;

    req->next=NULL;

```

```
cli ( ) ;
```

```
if (req->bh)
```

```
req->bh->b_dirt=0;
```

```
if ( ! (tmp=dev->current_request) ) { //只要硬盘空闲，就立即让  
当前请求项对应的缓冲块和硬盘进行交互
```

```
dev->current_request=req;
```

```
sti ( ) ;
```

```
(dev->request_fn) ( ) ;
```

```
return;
```

```
}
```

```
for ( ; tmp->next; tmp=tmp->next) //如果硬盘忙着，就把请求  
项载入请求项队列
```

```
if ( (IN_ORDER (tmp,req) ||
```

```
! IN_ORDER (tmp,tmp->next) ) &&
```

```
IN_ORDER (req,tmp->next) )
```

```
break;
```

```
req->next=tmp->next; //next指针用来组建队列
```

```
tmp->next=req;
```



```
sti ( ) ;  
  
}
```

`add_request`函数执行时会出现两种情况：如果硬盘是空闲的，就让硬盘处理当前请求项的请求；如果硬盘是忙的，即正在处理某个请求项，这时候又有请求项来了，就把这些请求项插入请求项队列，`next`指针用来组建请求项队列，情景如图7-5所示。

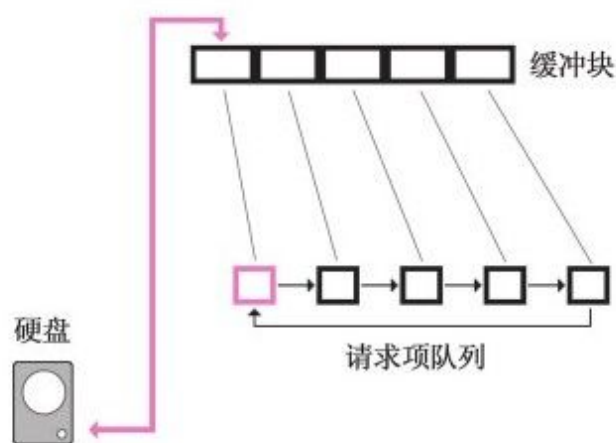


图 7-5 组建请求项队列的情景

我们再来看处理请求项队列中各项的代码：

```
//代码路径: kernel/blk_drv/hd.c:

static void read_intr (void)

{

if (win_result () ) {

bad_rw_intr () ;

do_hd_request () ;

return;

}

port_read (HD_DATA,CURRENT->buffer, 256) ;

CURRENT->errors=0;

CURRENT->buffer+=512;

CURRENT->sector++;

if (--CURRENT->nr_sectors) {

do_hd=&read_intr;

return;
```

```
}
```

```
end_request (1) ; //处理完一个请求项后处理善后工作
```

```
do_hd_request () ; //如果还有剩余请求项，继续下达交互指令；  
如果没有，就返回
```

```
}
```

```
static void write_intr (void)
```

```
{
```

```
if (win_result () ) {
```

```
bad_rw_intr () ;
```

```
do_hd_request () ;
```

```
return;
```

```
}
```

```
if (--CURRENT->nr_sectors) {
```

```
CURRENT->sector++;
```

```
CURRENT->buffer+=512;
```

```
do_hd=&write_intr;
```

```
port_write (HD_DATA,CURRENT->buffer, 256) ;
```

```
return;
```

```

}

end_request (1) ; //处理完一个请求项后处理善后工作

do_hd_request () ; //如果还有剩余请求项，继续下达交互指令；
如果没有，就返回

}

void do_hd_request (void)

{

int i,r;

unsigned int block,dev;

unsigned int sec,head,cyl;

unsigned int nsect;

INIT_REQUEST; //就在这里判断是否还有剩余的请求项

dev=MINOR (CURRENT-> dev) ;

block=CURRENT-> sector;

if (dev>=5*NR_HD||block+2>hd[dev].nr_sects) {

end_request (0) ;

goto repeat;

.....

```

```
}
```

//代码路径: kernel/blk_drv/hd.c:

```
extern inline void end_request (int uptodate)
```

```
{
```

```
.....
```

```
wake_up (&CURRENT->waiting) ;
```

```
wake_up (&wait_for_request) ;
```

```
CURRENT->dev=-1;
```

CURRENT=CURRENT->next; //将当前请求项设置为下一个, 为处理剩余请求项做准备

```
}
```

```
#define INIT_REQUEST\
```

```
repeat: \
```

```
if (! CURRENT) \//CURRENT为空, 说明没有剩余请求项了
```

```
return; \
```

```
if (MAJOR (CURRENT->dev) !=MAJOR_NR) \
```

```
panic (DEVICE_NAME": request list destroyed") ; \
```

```
if (CURRENT->bh) {\
```

```
if (! CURRENT->bh->b_lock) \
    panic (DEVICE_NAME": block not locked") ; \
}
```

从以上代码中可以看出，无论是读盘中断服务程序还是写盘中断服务程序，在执行完一对缓冲块和数据块的交互后，都要调用end_request

() 函数和do_hd_request () 函数，这样就形成了处理请求项队列中的请求项的循环操作。

do_hd_request () 函数中INIT_REQUEST这个宏，用以判断循环是否结束。如果当前请求项不为空，即队列中还有请求项对应的缓冲块需要交互，就继续下达交互命令，直到把请求项中所有的任务都执行完，即CURRENT为空，然后返回。此循环处理的情景如图7-6所示。

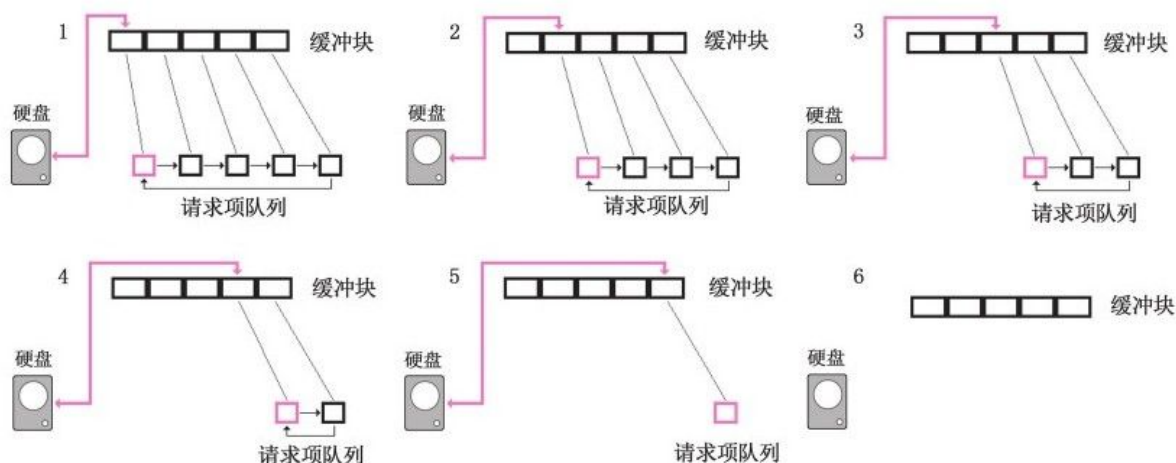


图 7-6 处理请求项队列的情景

请求项这样设计的目的只有一个，就是要尽快让缓冲块和硬盘数据块进行交互。

值得关注的是请求项的大小为32项，即 `request[32]`。为什么是32，而不是16或64呢？

这是因为数据在主机中交互的速度比在硬盘中交互的速度快2个数量级，也就是说，平均进程与缓冲区每交互100个缓冲块的数据，缓冲块与硬盘交互1个缓冲块的数据。主机中缓冲块的数量最

多为3000多块，缓冲块与请求项的数量之比正好是2个量级，与内存、硬盘交互速度之比匹配。如果请求项的数量过多，硬盘根本来不及处理，请求项就会闲置，浪费了内存；如果数量过少，由于没有足够的请求项，导致新的读写任务无法下达，硬盘空闲，而进程无合适的缓冲块可用，被频繁挂起，导致系统整体运行效率降低。而32这个数字，分寸拿捏得恰到好处。

7.4 uptodate和dirt的作用

前面一节介绍到，`b_dev`和`b_blocknr`两个字段是进程能共享缓冲块的基础，是缓冲块中数据是否仍然停留的标志。停留，就是要被共享，使用会延伸至两个方向：一个是进程方向，即进程能共享哪些缓冲块，不能共享哪些；一个是硬盘方向，即哪些需要同步到硬盘上，哪些不需要同步。而这两个使用方向的核心任务，就是确保缓冲块和数据块上数据的正确性。

`buffer_head`中`b_uptodate`和`b_dirt`这两个字段，都是为了解决缓冲块和数据块的数据正确性问题而存在的。

`b_uptodate`针对进程方向，它的作用是，告诉内核，只要缓冲块的`b_uptodate`字段被设置为1，缓冲块的数据已经是数据块中最新的，就可以放心地支持进程共享缓冲块的数据。反之，如果`b_uptodate`为0，就提醒内核缓冲块并没有用绑定的数据块中的数据更新，不支持进程共享该缓冲块。

`b_dirt`是针对硬盘方向的。只要缓冲块的`b_dirt`字段被设置为1，就是告诉内核，这个缓冲块中的内容已经被进程方向的数据改写了，最终需要同步到硬盘上。反之，如果为0，就不需要同步。

7.4.1 `b_uptodate`的作用

我们先来看进程方向上，如果没有b_uptodate字段的控制，会出现什么情况。

没有b_uptodate字段的控制，缓冲块与硬盘块绑定后，进程直接操作缓冲块中的数据，可能会出现错误。以读文件为例，如图7-7所示。

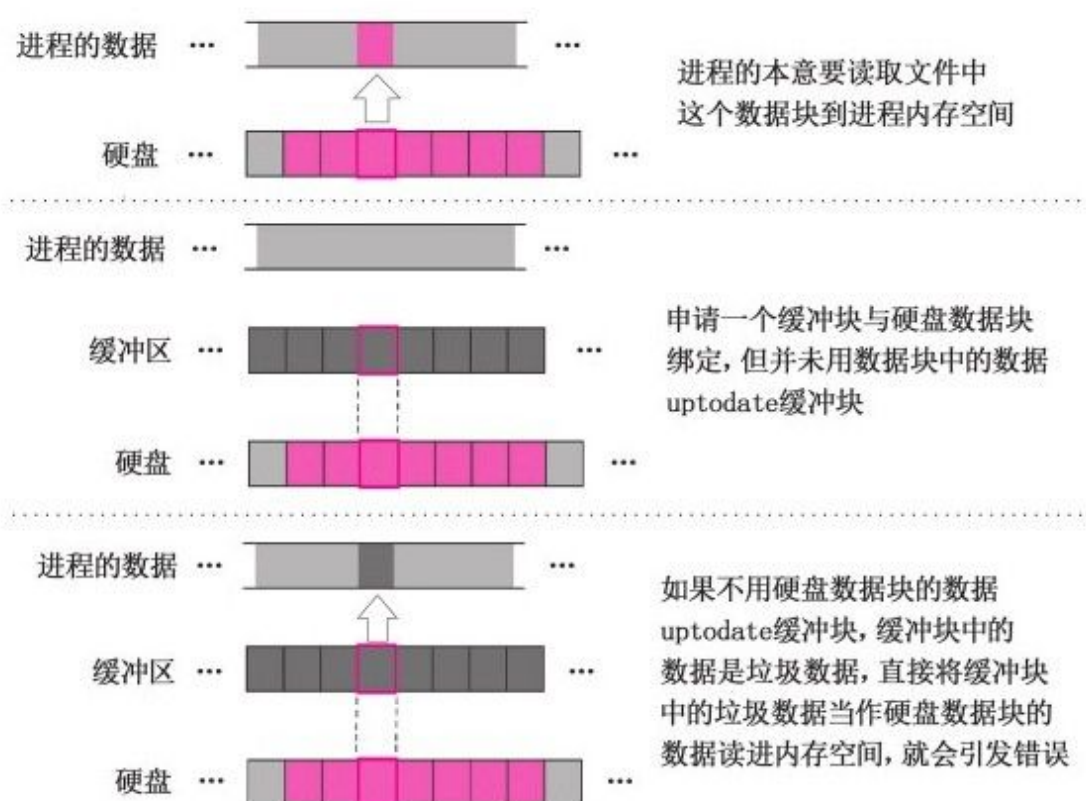


图 7-7 没有b_uptodate字段控制情况下读文件的情景

从图7-7中不难发现，由于缓冲块中的数据并未用数据块中的数据更新，b_uptodate为0，此时该缓冲块中是垃圾数据，所以读入进程的数据只是缓冲块中的垃圾数据，硬盘上的数据根本没读进来，违背了进程读取硬盘数据的本意，出现数据错误。

再比如写文件，如图7-8所示。

从图7-8中不难发现，进程原本要写入文件的数据量小于一个块，由于没有用数据块中的数据更新缓冲块，b_uptodate为0，同步数据时，缓冲块中的垃圾数据也写入了数据块，而且还覆盖了

原来的数据，引起数据错误，这也不是进程的本意。

可见，如果不用硬盘数据块中的数据更新缓冲块中的数据，后续的读文件、写文件对缓冲块的操作都不是建立在硬盘数据块中的数据的基础之上的，可能导致数据错误。设置**b_uptodate**为1，就标志着缓冲块中的数据是基于硬盘数据块的，内核可以放心地支持进程与缓冲块进行数据交互。

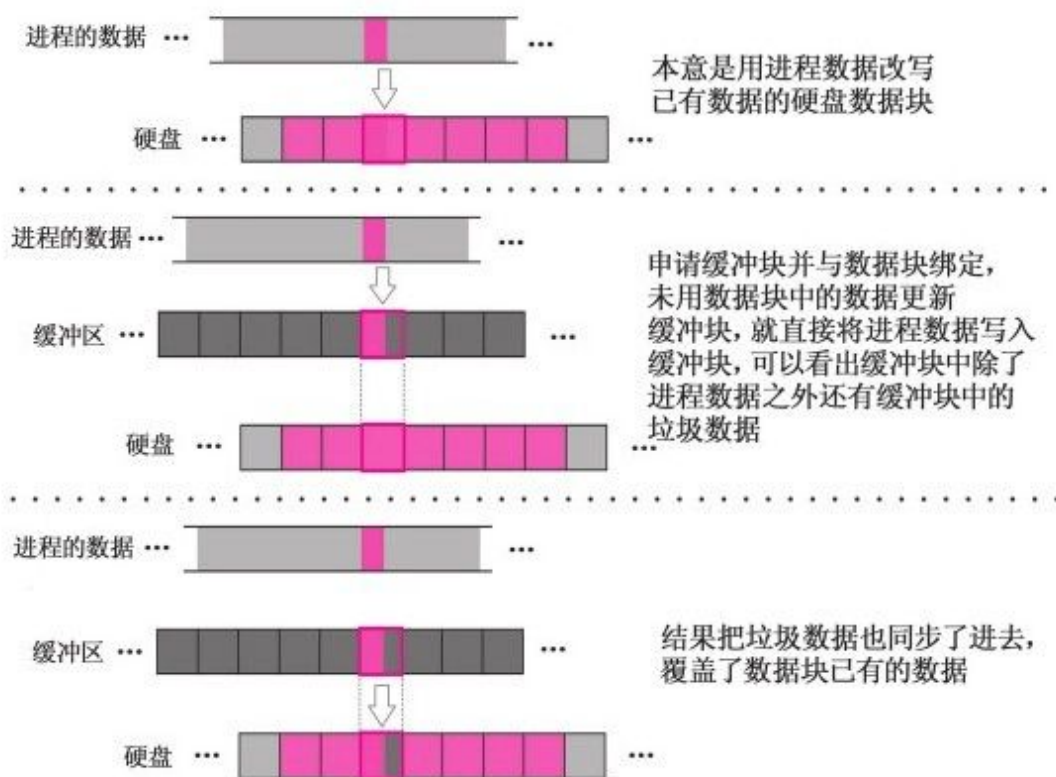


图 7-8 没有**b_uptodate**字段控制情况下写文件的情景

为此，当硬盘中断服务程序执行时，在数据从硬盘读入缓冲块后，或从缓冲块同步到硬盘后，都会将**b_uptodate**设置为1。代码如下：

//代码路径: `kernel/blk_drv/hd.c`:

```

static void read_intr (void) //读盘中断服务程序

{

.....

CURRENT->buffer+=512;

CURRENT->sector++;

if (--CURRENT->nr_sectors) {

do _hd=&read_intr;

return;

}

end_request (1) ; //处理完一个请求项后处理善后工作

do _hd_request ( ) ;

}

//代码路径: kernrl/blk_drv/hd.c:

static void write_intr (void) //写盘中断服务程序

{

.....

if (--CURRENT->nr_sectors) {

```

```
CURRENT->sector++;
```

```
CURRENT->buffer+=512;
```

```
do_hd=&write_intr;
```

```
port_write (HD_DATA,CURRENT->buffer, 256) ;
```

```
return;
```

```
}
```

```
end_request (1) ; //处理完一个请求项后处理善后工作
```

```
do_hd_request () ;
```

```
}
```

```
//代码路径: kernel/blk_drv/blk.h:
```

```
extern inline void end_request (int uptodate)
```

```
{
```

```
DEVICE_OFF (CURRENT->dev) ;
```

```
if (CURRENT->bh) {
```

CURRENT->bh->b_uptodate=uptodate; //将b_uptodate字段设置为1, 表示已经更新, 数据内容是同步的

```
unlock_buffer (CURRENT->bh) ;
```

```
}
```


.....

}

值得注意的是，`b_uptodate`被设置为1，是告诉内核，缓冲块中的数据已经用数据块中的数据更新过了，但并不等于两者的数据就完全一致。比如，为文件创建新数据块，就需要新建一个缓冲块与这个新建数据块确立绑定关系。关系确立后，先将缓冲块清零，然后将该缓冲块的**`b_uptodate`**字段设置为1。当然，此时并没有实质地同步数据，缓冲块和硬盘块的数据内容并不一致，但这并不影响数据的正确同步。

通过第5章的介绍我们得知，新建的数据块只可能有两个用途，要么用来存储文件的内容，要么用来存储文件的**`i_zone`**的间接块管理信息。

如果是存储文件内容，由于新建数据块和新建硬盘数据块，此时都是垃圾数据，都不是进程需要的，无所谓数据是否更新，结果“等效于”更新问题已经解决，所以将该缓冲块的**b_uptodate**字段设置为1（仔细想想，这时缓冲块不清零，也没有问题）。

如果是存储文件的间接块管理信息，必须清零，表示没有索引间接数据块，否则垃圾数据会导致索引错误，破坏文件操作的正确性。这时虽然缓冲块与硬盘数据块的数据不一致。依据相同的原理，**b_uptodate**字段设置为1仍然没问题。

设计者综合考虑，采取的策略是，只要为新建的数据块新申请了缓冲块，不管这个缓冲块将来用作什么，反正进程现在不需要里面的数据，

干脆全都清零。这样不管与之绑定的数据块用来存储什么信息，都无所谓，将该缓冲块的 `b_uptodate` 字段设置为1，更新问题“等效于”已解决。

代码如下：

```
//代码路径： fs/inode.c:

int create_block (struct m_inode * inode,int block) //创建一个新数据块
{
    return _bmap (inode,block, 1) ;
}

static int _bmap (struct m_inode * inode,int block,int create)
{
    .....

    if (block < 7) {
```

```

if (create && ! inode->i_zone[block])

if (inode->i_zone[block]=new_block (inode->i_dev) ) {

inode->i_ctime=CURRENT_TIME;

inode->i_dirt=1;

}

return inode->i_zone[block];

}

.....

}

```

//代码路径: fs/Bitmap.c:

```

int new_block (int dev) //在设备dev申请一个数据块

{

.....

if (bh->b_count != 1)

panic ("new block: count is != 1") ;

clear_block (bh->b_data) ;

bh->b_uptodate=1; //将该缓冲块设置为已更新

```

```
bh->b_dirt=1;  
  
brelse (bh) ;  
  
return j;  
  
}
```

`b_uptodate`被设置为1后，针对该缓冲块无非会发生读写两方面情况，下面我们看看会怎么样。

先看读的情况，缓冲块是新建的，虽然里面是垃圾数据，考虑到是新建文件，这时候不存在读没有内容的文件数据块的逻辑需求，内核代码不会做出这种愚蠢的动作。

再看写的情况，由于是新建缓冲块被清零、新建的硬盘数据块都是垃圾数据，此时缓冲块和数据块里面的数据都不是进程需要的，无所谓是

否更新、是否覆盖。无所谓更新，可以“等效地”看成已经更新。所以，执行写操作不会违背进程的本意。

我们来看图7-9。

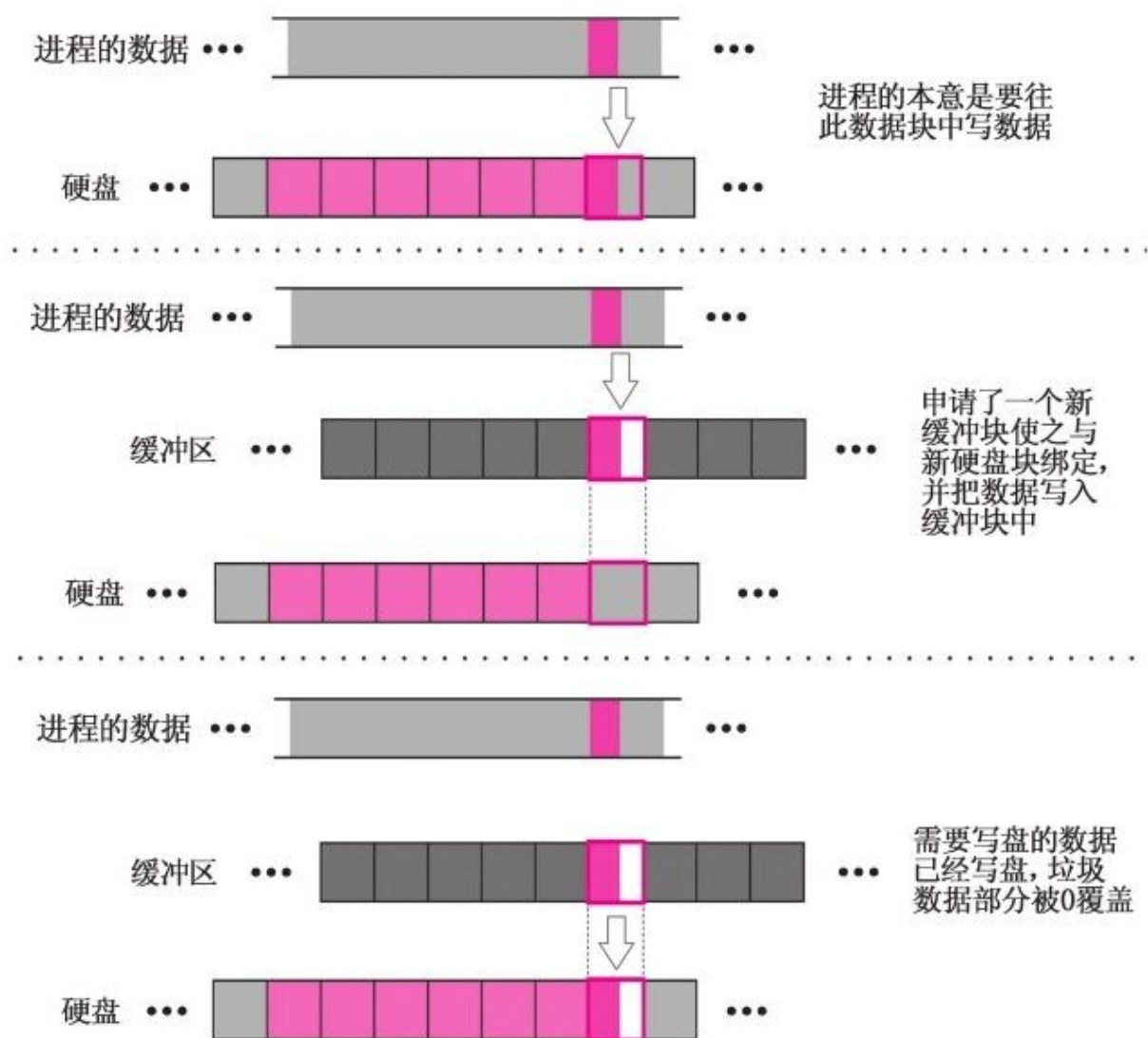


图 7-9 在**`b_uptodate`**字段控制情况下读文件的情景

以上就是内核中**`b_uptodate`**被设置为1的情景。图7-9表现了硬盘数据块用来存储文件内容的情景。

情景。白色部分代表清零了，不论存储的是文件的数据块数据还是间接块信息，都没有问题。

反之，如果缓冲区中数据没有更新，
b_uptodate为0，即更新问题没有解决，内核会阻拦进程，不让其共享缓冲块中的数据，无论读写，都不可以。其目的是为了避免前面讲述的没有更新带来的数据混乱。比如在读取块设备数据的时候，判断了两次，代码如下：

```
//代码路径： fs/Buffer.c:

struct buffer_head * bread (int dev,int block)

{

    struct buffer_head * bh;

    if ( ! (bh=getblk (dev,block) ) )

        panic ("bread:  getblk returned NULL\n") ;
```


if (bh->b_uptodate) //申请到缓冲块后，先看是否已更新，以此决定是否返回使用该缓冲块

```
return bh;
```

```
ll_rw_block (READ,bh) ;
```

```
wait_on_buffer (bh) ;
```

if (bh->b_uptodate) //从硬盘读进内容后，再次检查是否已更新，以此决定是否返回使用该缓冲块

```
return bh;
```

```
brelse (bh) ;
```

```
return NULL;
```

```
}
```

此代码中，getblk函数很有可能在缓冲区中找到一个已经建立了绑定关系（b_dev和b_blocknr都匹配），而且正好可以被当前进程使用的缓冲块，但就是由于b_uptodate为0，这个缓冲块也不能使用，只好再释放掉。

再比如，为与文件中已有的数据块交互而新申请的一个缓冲块，就把b_uptodate标志设置为0，表示此缓冲块数据现在还没有被更新，不能被进程共享，代码如下：

//代码路径： fs/Buffer.c:

```
struct buffer_head * getblk (int dev,int block)
```

```
{
```

```
.....
```

```
if (find_buffer (dev,block) )
```

```
goto repeat;
```

```
bh->b_count=1;
```

```
bh->b_dirt=0;
```

```
bh->b_uptodate=0; //数据还没有被更新，还不能被进程共享
```

```
remove_from_queues (bh) ;
```

```
bh->b_dev=dev;
```

```
bh->b_blocknr=block;
```

```
insert_into_queues (bh) ;  
  
return bh;  
  
}
```

7.1 节中介绍过这部分代码，一个新的缓冲块就是在这里诞生的，刚诞生的一刻，数据肯定和硬盘数据块是不一致的，所以要把**b_uptodate**设置为0，确保不被进程误用。

7.4.2 b_dirt的作用

b_uptodate标志设置为1后，内核就可以支持进程共享该缓冲块的数据了，读写都可以。读操作不会改变缓冲块中数据的内容，所以不影响数据；而执行了写操作后，就改变了缓冲块数据内容，就要将b_dirt标志设置为1。比如，往块设备文件写入数据，往普通文件写入数据，等等。具体代码如下：

//代码路径：fs/blk_dev.c:

```
int block_write (int dev,long * pos,char * buf,int count) //块设备文件内容写入缓冲块
```

```
{
```

```
.....
```

```
offset=0;
```

```
*pos+=chars;
```

```
written+=chars;
```

```
count-=chars;
```

```
while (chars-->0)
```

```
* (p++) =get_fs_byte (buf++) ;
```

```
bh->b_dirt=1;
```

```
brelse (bh) ;
```

```
.....
```

```
}
```

```
//代码路径: fs/file_dev.c:
```

```
int file_write (struct m_inode * inode,struct file * filp,char * buf,int  
count) //普通文件内容写入缓冲块
```

```
{
```

```
.....
```

```
c=pos%BLOCK_SIZE;
```

```
p=c+bh->b_data;
```

```
bh->b_dirt=1;
```

```

c=BLOCK_SIZE-c;

if (c>count-i) c=count-i;

pos+=c;

if (pos>inode->i_size) {

inode->i_size=pos;

inode->i_dirt=1;

}

i+=c;

while (c-->0)

* (p++) =get_fs_byte (buf++) ;

.....

}

```

//代码路径: fs/file_dev.c:

```
static struct buffer_head * add_entry (struct m_inode * dir,
```

```
const char * name,int namelen,struct dir_entry ** res_dir) //目录文件
需要加载目录项, 用到写缓冲块
```

```
{
```

```
if (i * sizeof (struct dir_entry) >= dir->i_size) {

de->inode=0;

dir->i_size= (i+1) *sizeof (struct dir_entry) ;

dir->i_dirt=1;

dir->i_ctime=CURRENT_TIME;

}

if (! de->inode) {

dir->i_mtime=CURRENT_TIME;

for (i=0; i<NAME_LEN; i++)

de->name[i]= (i<namelen) ?get_fs_byte (name+i) : 0;

bh->b_dirt=1;

*res_dir=de;

return bh;

}

}
```

改变了缓冲块数据内容，b_uptodate标志要不要因此重新设置为0，而禁止内核继续支持进程共享该缓冲块呢？我们来看图7-10。

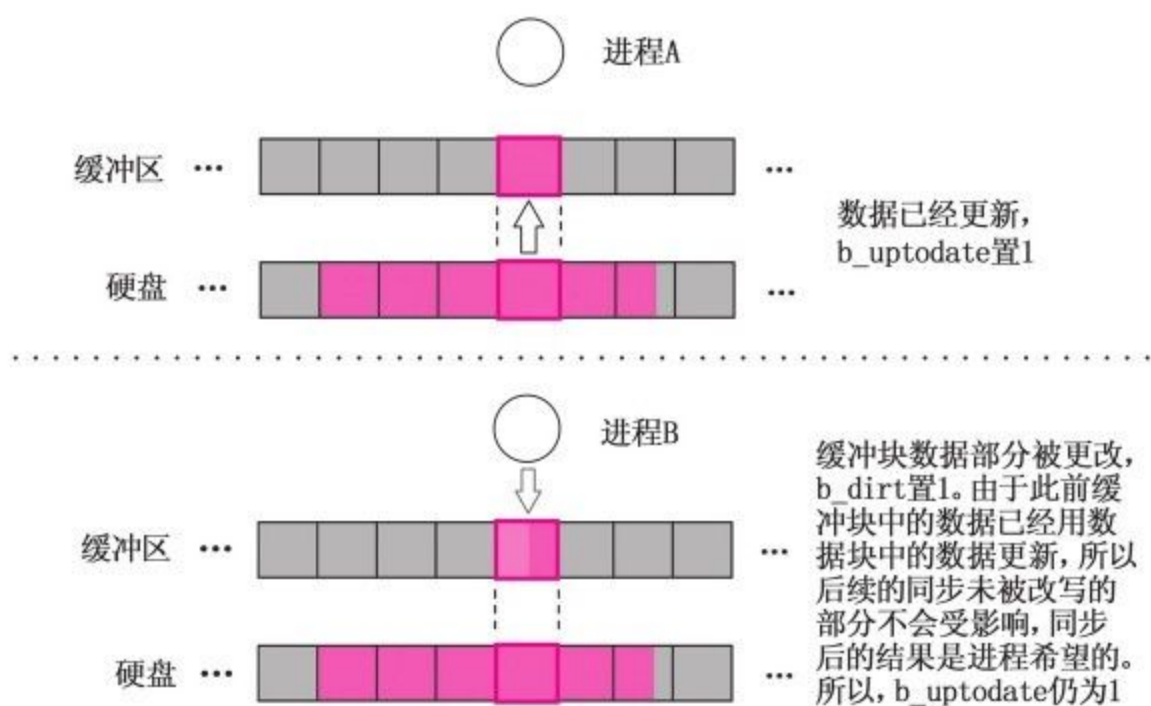


图 7-10 往缓冲块中写入数据的情景

从图7-10中不难发现，由于这个缓冲块中的数据此前已经用硬盘数据块中的数据更新过，所以，往缓冲块中写入新数据后，缓冲块中没有写

入新数据的部分仍与硬盘数据块对应的数据部分相同，将来往数据块上同步的时候，所有的数据都是进程希望同步到硬盘数据块上的，不会把垃圾数据同步到数据块中。所以**b_uptodate**仍然是1，不需要改变，这个缓冲块中的数据仍然可以被进程共享，继续读写都没有问题。我们来看继续写入数据的情景（见图7-11）。

以此类推，不断地往缓冲块写入数据，缓冲区中的数据自然而然地不断变化，将来同步的时候，这些新数据自然会像进程希望的那样同步到硬盘数据块中。

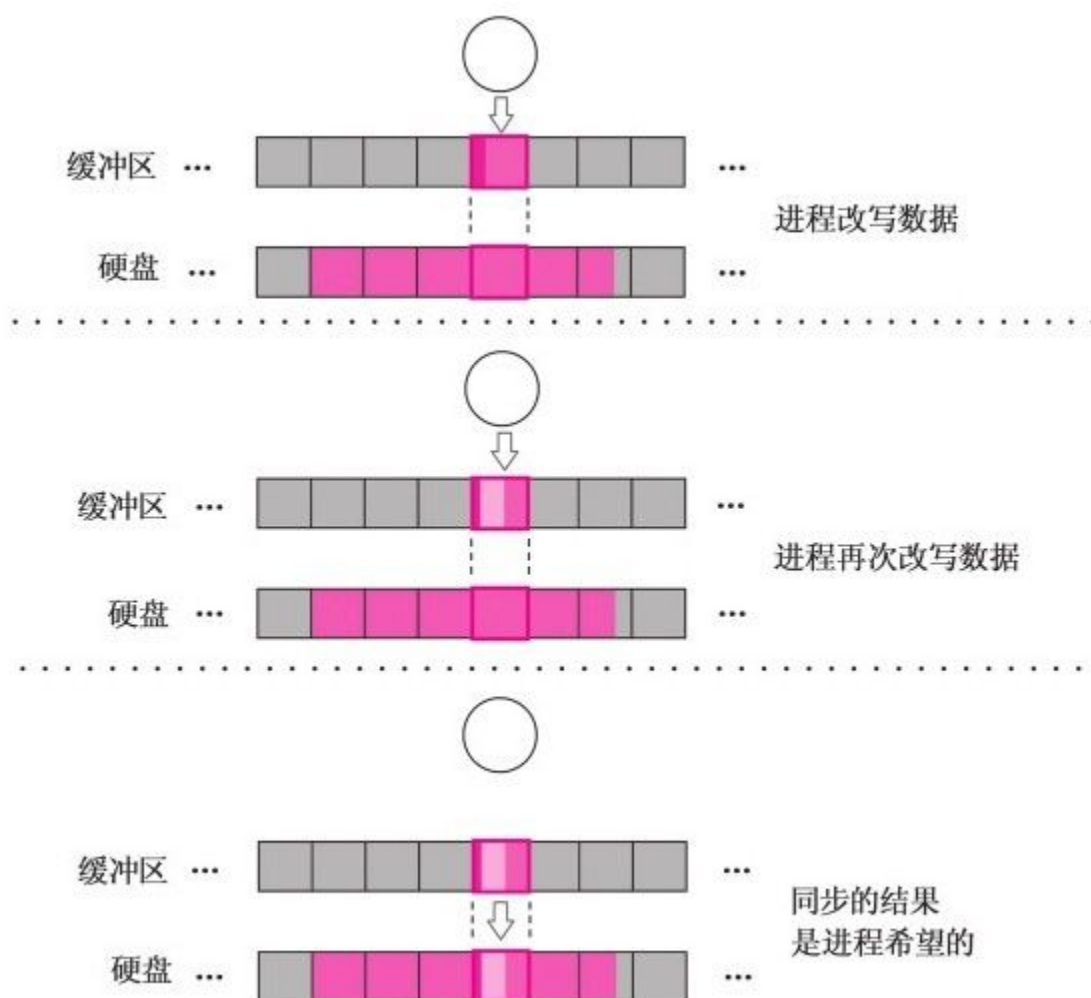


图 7-11 继续往缓冲块中写入数据的情景

buffer_head中在进程方向和硬盘方向分别设置了两个字段（b_uptodate和b_dirt），请求项结构中也有这两个方向上的考虑。相比写操作而

言，读操作对用户进程更紧迫，所以请求项为这两种操作设定了大小不同的空间，代码如下：

//代码路径： kernel/blk_drv/ll_rw_blk.c:

```
static void make_request (int major,int rw,struct buffer_head * bh)

{

.....

lock_buffer (bh) ;

    if ( (rw==WRITE&&! bh->b_dirt) || (rw==READ&&bh->
b_uptodate) ) {

        unlock_buffer (bh) ;

        return;

    }

repeat:

    if (rw==READ)

        req=request+NR_REQUEST;

    else
```

```
req=request+ ( (NR_REQUEST*2) /3) ;  
  
while (--req >=request)  
  
if (req->dev<0)  
  
break;  
  
.....  
  
}
```

从以上代码不难发现，request[32]中只有2/3的空间可以用来写操作，而全部的空间都可以用来读操作，在同等的条件下，读操作执行的机会更多。

另外，仔细考察b_uptodate、b_dirt可以发现，只要b_uptodate被设置为1，进程就可以共享里面的数据。而且在缓冲区中没有进程可以直接共享的缓冲块的情况下，只要b_count为0，就可

以挪作他用，让该缓冲块与其他数据块绑定，另行使用，不用担心数据会发生错误。但如果b_dirt被设置为1，情况就不一样了，这个缓冲块的数据已经和数据块上的不一致了，需要同步，虽然用不着立即就同步，但在同步之前不能挪作他用，否则就会覆盖掉这些数据，硬盘数据块中的数据没有体现进程改写的内容，出现数据混乱。这时如果出现缓冲块不够进程用的情况，那就只好让进程等待，等同步完成后，内核就会立刻将b_dirt设置为0，腾出更多缓冲块供进程使用。代码如下：

```
//代码路径: kernel/blk_drv/ll_rw_blk.c:

static void add_request (struct blk_dev_struct * dev, struct request *
req)

{
```

```
struct request * tmp;

req->next=NULL;

cli ( ) ;

if (req->bh)

req->bh->b_dirt=0;

if ( ! (tmp=dev->current_request) ) {

dev->current_request=req;

sti ( ) ;

(dev->request_fn) ( ) ;

return;

}

.....

}
```

7.4.3 i_uptodate、i_dirt和s_dirt的作用

以上介绍了控制文件内容正确性的方面，内容通过b_uptodate和b_dirt这两个字段，保证缓冲区数据与硬盘数据块数据的正确性。文件管理信息也有类似的字段，比如inode_table[32]中存储的i节点，在多进程操作同一文件时，就要共享文件i节点信息。为此它的数据结构中也设计了两个字段：i_uptodate（Linux 0.11中没有实际使用）和i_dirt。代码如下：

```
//代码路径：include/linux/fs.h:
```

```
struct m_inode{  
  
    unsigned short i_mode;  
  
    unsigned short i_uid;
```

```
unsigned long i_size;

unsigned long i_mtime;

unsigned char i_gid;

unsigned char i_nlinks;

unsigned short i_zone[9];

/*these are in memory also*/

struct task_struct * i_wait;

unsigned long i_atime;

unsigned long i_ctime;

unsigned short i_dev;

unsigned short i_num;

unsigned short i_count;

unsigned char i_lock;

unsigned char i_dirt;

unsigned char i_pipe;

unsigned char i_mount;

unsigned char i_seek;
```



```
unsigned char i_update;  
  
};
```

设计i_dirt字段不难理解，比如改变文件大小后，文件i节点中就要改变对大小的记录，这样inode_table[32]中的i节点和硬盘上的内容就不一样了，需要同步。i节点中i_uptodate标志并没有在系统中用到过。这是因为，这些文件管理信息在硬盘上都是以数据块的形式存储的，它们也都是以块的形式载入缓冲区的，载入缓冲区后与硬盘数据块内容一样，等价于已经更新，直接可以用来共享，不需要在管理结构中再搞一套i_uptodate标志了。

super_block[8]中存储的超级块，也存在被进程共享的问题。超级块中保存着整个文件系统的

管理信息，多进程操作文件时，免不了都会用到。它的数据结构中也有一个字段：s_dirt，代码如下：

```
//代码路径：include/linux/fs.h:
```

```
struct super_block{

    unsigned short s_ninodes;

    unsigned short s_nzones;

    unsigned short s_imap_blocks;

    unsigned short s_zmap_blocks;

    unsigned short s_firstdatazone;

    unsigned short s_log_zone_size;

    unsigned long s_max_size;

    unsigned short s_magic;

    /*These are only in memory*/

    struct buffer_head * s_imap[8];
```

```
struct buffer_head * s_zmap[8];

unsigned short s_dev;

struct m_inode * s_isup;

struct m_inode * s_imount;

unsigned long s_time;

struct task_struct * s_wait;

unsigned char s_lock;

unsigned char s_rd_only;

unsigned char s_dirt;

};
```

结构中没有类似uptodate这样的字段，理由和i节点结构中i_uptodate没有被用到是一样的。它们也都是以块的形式载入缓冲区的，载入缓冲区后与硬盘数据块内容一样，等价于已经更新，不需要在管理结构中再搞一套uptodate标志了。而

s_dirt字段，除了在读超级块时被设置为0后，再没有被使用过。这是因为，在Linux 0.11中，进程共享超级块信息，全部从super_block[8]中读取信息，并没有往表项中写入数据，所以没有s_dirt字段。

7.5 count、lock、wait、request的作用

数据停留在缓冲块后，在进程方向的使用问题继续延伸，就是本节要介绍的b_count、b_lock和*b_wait。

7.5.1 b_count的作用

进程向内核提出申请的时候，内核只能在下面两种情况中做出选择：让进程和其他进程共享某个缓冲块，该缓冲块的所有控制字段的数值也一并先继承下来；为进程申请一个没被任何进程占用的缓冲块，所有的控制字段重新设置。

要做选择，进程就要知道哪些缓冲块已经被其他进程占用了，哪些没有被占用。有些缓冲块可能不止被一个进程共享，这就需要在缓冲块中设置一个字段，使内核能够随时知道“每个缓冲块有多少进程在共享”，这个字段就是**b_count**。

缓冲区在初始化的时候，没有进程引用缓冲块，所以每个缓冲块的**b_count**被设置为0。代码如下：

//代码路径： fs/buffer.c:

```
void buffer_init (long buffer_end)

{

.....

h->b_dev=0;

h->b_dirt=0;
```

```
h->b_count=0; //没有进程引用缓冲块，引用计数为0

h->b_lock=0;

h->b_uptodate=0;

h->b_wait=NULL;

h->b_next=NULL;

h->b_prev=NULL;

h->b_data= (char *) b;

h->b_prev_free=h-1;

h->b_next_free=h+1;

.....

}
```

新申请一个缓冲块时，这个缓冲块必须没有被任何进程占用，**b_count**值设置为0；申请到后，缓冲块被第一个进程共享，**b_count**被设置为1。代码如下：

//代码路径: fs/buffer.c:

```
struct buffer_head * getblk (int dev,int block)
```

```
{
```

```
.....
```

```
tmp=free_list;
```

```
do{
```

```
if (tmp->b_count) //引用计数必须为0
```

```
continue;
```

```
if (! bh||BADNESS (tmp) < BADNESS (bh) ) {//才能考虑重  
新申请
```

```
bh=tmp;
```

```
if (! BADNESS (tmp) )
```

```
break;
```

```
}
```

```
/*and repeat until we find something good*/
```

```
}while ( (tmp=tmp->b_next_free) !=free_list) ;
```

```
.....
```


bh->b_count=1; //新缓冲块，意味着只有当前进程在引用它，所以b_count被设置为1

 bh->b_dirt=0;

 bh->b_uptodate=0;

 remove_from_queues (bh) ;

 bh->b_dev=dev;

 bh->b_blocknr=block;

 insert_into_queues (bh) ;

 return bh;

}

缓冲块陆续被更多的进程共享，b_count的数值在原来的基础上逐渐累加。代码如下：

//代码路径： fs/buffer.c:

struct buffer_head * getblk (int dev,int block)

{

```

    struct buffer_head * tmp, *bh;

    repeat:

        if (bh=get_hash_table (dev,block) ) //遍历哈希表，看能不能和
        其他进程共享缓冲块

        return bh;

        .....

    }

    struct buffer_head * get_hash_table (int dev,int block)

    {

        struct buffer_head * bh;

        for ( ; ) {

            if ( ! (bh=find_buffer (dev,block) ) )

                return NULL;

            bh->b_count++; //如果发现可以共享，则该缓冲块又多了一个进
            程引用它， b_count递增

            wait_on_buffer (bh) ;

            if (bh->b_dev==dev&&bh->b_blocknr==block)

                return bh;

```

```
bh->b_count--;  
  
}  
  
}
```

而在进程读写文件完毕后，不再需要共享缓冲块，内核会解除该进程和缓冲块的关系，b_count数值随之减1。如果所有进程和该缓冲块的关系都解除了，则b_count的值就被递减为0，这个缓冲块就又可以被当做新缓冲块来申请了。代码如下：

```
//代码路径： fs/file_dev.c:  
  
int file_read (struct m_inode * inode,struct file * filp,char * buf,int  
count) //读文件  
  
{  
  
.....  
  
while (chars-->0)
```

```
put_fs_byte (* (p++) , buf++) ;
```

```
brelse (bh) ; //递减引用计数
```

```
}else{
```

```
while (chars-->0)
```

```
put_fs_byte (0, buf++) ;
```

```
.....
```

```
}
```

```
int file_write (struct m_inode * inode,struct file * filp,char * buf,int  
count) //写文件
```

```
{
```

```
.....
```

```
while (c-->0)
```

```
* (p++) =get_fs_byte (buf++) ;
```

```
brelse (bh) ; //递减引用计数
```

```
}
```

```
inode->i_mtime=CURRENT_TIME;
```

```
if (! (filp->f_flags&O_APPEND) ) {
```

```

filp->f_pos=pos;

inode->i_ctime=CURRENT_TIME;

}

.....

}

//代码路径: fs/buffer.c:

void brelse (struct buffer_head * buf)

{

if (! buf)

return;

wait_on_buffer (buf) ;

if (! (buf->b_count--))

panic ("Trying to free free buffer") ;

wake_up (&buffer_wait) ;

}

```

值得注意的是，在所有共享缓冲块的进程全部脱离共享关系后，虽然**b_count**肯定为0，但这并不等于缓冲块与数据块解除了绑定关系。如果将来某个进程再操作这个缓冲块，只要这个缓冲块的**b_dev**、**b_blocknr**没有改变，就不需要从硬盘中重新读取，完全可以直接沿用这个缓冲块。

7.5.2 i_count的作用

进程与缓冲块之间共享的是文件的内容数据，不仅管理文件的内容数据时需要b_count字段，而且文件的管理信息中，凡是需要“搜索空闲项”、“搜索到空闲项后可以另作他用”的数据结构，都需要与之类似的字段。比如inode_table[32]中，就有类似字段。代码如下：

//代码路径：include/linux/fs.h:

```
struct m_inode{  
  
    unsigned short i_mode;  
  
    unsigned short i_uid;  
  
    unsigned long i_size;  
  
    unsigned long i_mtime;  
  
    unsigned char i_gid;
```

```
unsigned char i_nlinks;

unsigned short i_zone[9];

/*these are in memory also*/

struct task_struct * i_wait;

unsigned long i_atime;

unsigned long i_ctime;

unsigned short i_dev;

unsigned short i_num;

unsigned short i_count;

unsigned char i_lock;

unsigned char i_dirt;

unsigned char i_pipe;

unsigned char i_mount;

unsigned char i_seek;

unsigned char i_update;

};
```

`inode_table[32]`是文件管理信息，进程引用了文件数据块所对应的缓冲块，也就必然相应地引用了`inode_table[32]`中的i节点项，此两者是同步的。所以`inode_table[32]`中也需要`i_count`这一字段来标识该i节点项被多少进程共享了。如果没被共享，则`i_count`就是0，就可以被当做空闲项。比如进程要打开一个从未打开的文件，无法与其他进程共享i节点项时，就可以用这个空闲i节点项来装载i节点。

而`super_block[8]`则与此不同，一个设备就一个超级块项目，整个系统就只能安装8个超级块，这都是有数的。一个超级块项从加载到文件系统卸载，只代表某个设备的超级块，所以不存在是否空闲、是否打算着另作他用的情况。多个进程

可以加载相同的文件系统，需要操作相同的超级块，但就用不着count这样的字段来记录该超级块被引用的次数。代码如下：

```
//代码路径: include/linux/fs.h:
```

```
struct super_block{
```

```
    unsigned short s_ninodes;
```

```
    unsigned short s_nzones;
```

```
    unsigned short s_imap_blocks;
```

```
    unsigned short s_zmap_blocks;
```

```
    unsigned short s_firstdatazone;
```

```
    unsigned short s_log_zone_size;
```

```
    unsigned long s_max_size;
```

```
    unsigned short s_magic;
```

```
    /*These are only in memory*/
```

```
    struct buffer_head * s_imap[8];
```

```
struct buffer_head * s_zmap[8];

unsigned short s_dev;

struct m_inode * s_isup;

struct m_inode * s_imount;

unsigned long s_time;

struct task_struct * s_wait;

unsigned char s_lock;

unsigned char s_rd_only;

unsigned char s_dirt;

};
```

从以上代码中可以看出，没有类似count的字段。

值得注意的是，除了i节点和超级块外，文件管理信息还包括i节点位图和逻辑块位图。这两类文件管理信息并没有专用的数据结构，但它们也

要支持共享。它们存储在缓冲块中，而且是常驻。这些缓冲块只为i节点位图、逻辑块位图使用。代码如下：

```
//代码路径: fs/super.c:

static struct super_block * read_super (int dev)

{

.....

for (i=0; i<s->s_imap_blocks; i++) //往缓冲块中载入i节点位图

if (s->s_imap[i]=bread (dev,block) )

block++;

else

break;

for (i=0; i<s->s_zmap_blocks; i++) //往缓冲块中载入逻辑块位图

if (s->s_zmap[i]=bread (dev,block) )

block++;
```

else

break;

if (block != 2 + s->s_imap_blocks + s->s_zmap_blocks) { //如果出现异常情况，再释放

for (i=0; i<I_MAP_SLOTS; i++)

brlse (s->s_imap[i]) ;

for (i=0; i<Z_MAP_SLOTS; i++)

brlse (s->s_zmap[i]) ;

s->s_dev=0;

free _super (s) ;

return NULL;

}

s->s_imap[0]->b_data[0]=1;

s->s_zmap[0]->b_data[0]=1;

free _super (s) ;

return s;

}

//代码路径: fs/buffer.c:

```
struct buffer_head * bread (int dev,int block) //读取底层块设备数据
```

```
{
```

```
struct buffer_head * bh;
```

```
    if ( ! (bh=getblk (dev,block) ) ) //申请缓冲块时要用到设备号  
和块号
```

```
panic ("bread: getblk returned NULL\n") ;
```

```
if (bh->b_uptodate)
```

```
return bh;
```

```
.....
```

```
}
```

```
struct buffer_head * getblk (int dev,int block)
```

```
{
```

```
.....
```

```
if (find_buffer (dev,block) )
```

```
goto repeat;
```

```
bh->b_count=1; //引用计数为1
```

```
bh->b_dirt=0;  
  
bh->b_uptodate=0;  
  
.....  
  
}
```

从以上代码中可以看出，i节点位图、超级块位图载入缓冲块后，这些缓冲块的**b_count**被设置为1，之后并没有将其释放过，这样这些缓冲块的引用计数就无法还原为0了。所以任何进程申请新缓冲块的时候，都无法申请到它们，所以这些缓冲块成为专用。

7.5.3 b_lock、*b_wait的作用

内核为进程申请到缓冲块，尤其是申请到b_count为0的缓冲块时，因为同步的原因，有可能这个缓冲块正在与硬盘交互数据，为此buffer_head结构中设置了b_lock字段。如果此字段被设置为1，就说明正在和硬盘交互数据，内核就会拦截进程对该缓冲块的操作，等到与硬盘的交互结束时，再把该字段设置为0，以此解除对进程方面的拦截。

如果为进程申请到的缓冲块中b_lock字段被设置为1，即便已经申请到了，该进程也需要挂起，直到该缓冲块被解锁后，才能访问。在缓冲块被加锁的过程中，而且无论有多少进程申请到

了这个缓冲块，都不能立即操作该缓冲块，都要挂起，并切换到其他进程去执行。这就需要记录有哪些进程因为等待这个缓冲块的解锁而被挂起了。由于使用了进程等待队列，所以一个字段就可以解决这个记录问题。这个字段就是***b_wait**。

这两个字段往往是联合使用的，我们来看如下代码。

在急速以前，初始化缓冲块的时候，**b_lock**全部被设置为0，***b_wait**被设置为**NULL**。

//代码路径: fs/buffer.c:

```
void buffer_init (long buffer_end)

{

.....

h->b_dev=0;
```

```
h->b_dirt=0;

h->b_count=0;

h->b_lock=0;

h->b_uptodate=0;

h->b_wait=NULL;

h->b_next=NULL;

h->b_prev=NULL;

h->b_data= (char *) b;

h->b_prev_free=h-1;

h->b_next_free=h+1;

.....

}
```

缓冲块被申请后，开始底层块操作前，就要先把该块加锁，即把**b_lock**设置为1，然后进行底层操作。执行代码如下：

//代码路径: kernel/blk_drv/ll_rw_block.c:

```
static void make_request (int major,int rw,struct buffer_head * bh)

{

.....

if (rw!=READ&&rw!=WRITE)

panic ("Bad block dev command,must be R/W/RA/WA") ;

lock_buffer (bh) ; //给缓冲块加锁

if ( (rw==WRITE&&! bh->b_dirt) || (rw==READ&&bh->
b_uptodate) ) {

unlock_buffer (bh) ;

return;

}

.....

}

static inline void lock_buffer (struct buffer_head * bh)

{

cli () ;
```

```
while (bh->b_lock) //如果缓冲块已经加锁

sleep_on (&bh->b_wait) ; //直接将进程挂起

bh->b_lock=1; //给缓冲块加锁

sti () ;

}
```

缓冲块与硬盘数据块开始交互数据时，
`lock_buffer ()` 函数先判断缓冲块是否加锁。如果加锁了（很有可能该缓冲块早就被别的进程申请了，现在正与硬盘交互数据），就直接调用 `sleep_on ()` 函数将进程挂起，并切换到其他进程去执行。等到将来切换回当前进程后，再将缓冲块继续加锁。如果没加锁，就将其加锁，以防其他进程误操作。`b_lock`和`*b_wait`的联用不仅体现在这里，其他只要需要判断缓冲块的使用状态的，都需要两者的联用。代码如下：

//代码路径: fs/buffer.c:

```
struct buffer_head * bread (int dev,int block)

{

struct buffer_head * bh;

if ( ! (bh=getblk (dev,block) ) )

panic ("bread: getblk returned NULL\n") ;

if (bh->b_uptodate)

return bh;

ll_rw_block (READ,bh) ;

wait_on_buffer (bh) ; //检测进程是否需要等待缓冲块解锁

if (bh->b_uptodate)

return bh;

brelse (bh) ;

return NULL;

}

static inline void lock_buffer (struct buffer_head * bh)
```

```
{  
  
cli ();  
  
while (bh->b_lock) //如果缓冲块已经加锁  
  
sleep_on (&bh->b_wait) ; //直接将进程挂起  
  
bh->b_lock=1; //给缓冲块加锁  
  
sti ();  
  
}
```

给缓冲块加锁并将进程挂起时，两者联用；相反，给缓冲块解锁并将进程唤醒时，两者也是联用。代码如下：

//代码路径： kernel/blk_drv/ll_rw_block.c:

```
static void make_request (int major,int rw,struct buffer_head * bh)  
  
{  
  
.....  
  
lock_buffer (bh) ;
```

```
    if ( (rw==WRITE && ! bh->b_dirt) || (rw==READ && bh->
b_uptodate) ) {
```

```
        unlock_buffer (bh) ; //给缓冲块解锁并唤醒进程
```

```
    return;
```

```
}
```

```
if (rw==READ)
```

```
    req=request+NR_REQUEST;
```

```
.....
```

```
}
```

```
static inline void unlock_buffer (struct buffer_head * bh) //给缓冲块
解锁并唤醒进程
```

```
{
```

```
    if (! bh->b_lock)
```

```
        printk ("ll_rw_block.c: buffer not locked\n\r") ;
```

```
    bh->b_lock=0;
```

```
    wake_up (&bh->b_wait) ;
```

```
}
```

读盘或写盘结束后，中断服务程序执行，会给缓冲块解锁，随后也将原来等待该缓冲块的进程唤醒。代码如下：

//代码路径: kernel/blk_drv/blk.h:

extern inline void end_request (int uptodate) //处理请求项操作完毕后的善后工作

{

DEVICE _OFF (CURRENT->dev) ;

if (CURRENT->bh) {

CURRENT->bh->b_uptodate=uptodate;

unlock_buffer (CURRENT->bh) ; //给缓冲块解锁并唤醒进程

}

if (! uptodate) {

printk (DEVICE_NAME"I/O error\n\r") ;

printk ("dev%04x,block%d\n\r", CURRENT->dev,

CURRENT->bh->b_blocknr) ;


```
}
```

```
.....
```

```
}
```

```
static inline void unlock_buffer (struct buffer_head * bh) //给缓冲块  
解锁并唤醒进程
```

```
{
```

```
if (! bh->b_lock)
```

```
printk (DEVICE_NAME": free buffer being unlocked\n") ;
```

```
bh->b_lock=0; //给缓冲块解锁
```

```
wake_up (&bh->b_wait) ; //唤醒等待缓冲块的进程
```

```
}
```

7.5.4 i_lock、i_wait、s_lock、*s_wait的作用

在共享文件内容时，b_lock和*b_wait字段存在于缓冲块中，共享文件管理信息和共享文件内容是配套的，所以，文件管理信息的数据结构也有同样类似的字段存在，比如inode_table[32]，super_block[8]。代码如下：

```
//代码路径：include/linux/fs.h:
```

```
struct m_inode{

unsigned short i_mode;

unsigned short i_uid;

unsigned long i_size;

unsigned long i_mtime;
```

```
unsigned char i_gid;

unsigned char i_nlinks;

unsigned short i_zone[9];

/*these are in memory also*/

struct task_struct * i_wait;

unsigned long i_atime;

unsigned long i_ctime;

unsigned short i_dev;

unsigned short i_num;

unsigned short i_count;

unsigned char i_lock;

unsigned char i_dirt;

unsigned char i_pipe;

unsigned char i_mount;

unsigned char i_seek;

unsigned char i_update;

};
```

```
struct super_block{

unsigned short s_ninodes;

unsigned short s_nzones;

unsigned short s_imap_blocks;

unsigned short s_zmap_blocks;

unsigned short s_firstdatazone;

unsigned short s_log_zone_size;

unsigned long s_max_size;

unsigned short s_magic;

/*These are only in memory*/

struct buffer_head * s_imap[8];

struct buffer_head * s_zmap[8];

unsigned short s_dev;

struct m_inode * s_isup;

struct m_inode * s_imount;

unsigned long s_time;

struct task_struct * s_wait;
```

```
unsigned char s_lock;  
  
unsigned char s_rd_only;  
  
unsigned char s_dirt;  
  
};
```

类似lock和wait的字段，不仅在文件管理信息中都存在，而且，使用的时候，也是联用的，因为它们也要为共享服务。

inode_table[32]中i_lock和i_wait联用时的代码如下：

```
//代码路径： fs/inode.c:  
  
static void read_inode (struct m_inode * inode) //读i节点  
{  
  
.....  
  
lock_inode (inode) ; //给i节点加锁
```

```

if ( ! (sb=get_super (inode->i_dev) ) )

panic ("trying to read inode without dev" ) ;

.....

* (struct d_inode *) inode=

( (struct d_inode *) bh->b_data)

[ (inode->i_num-1) %INODES_PER_BLOCK];

brelse (bh) ;

unlock_inode (inode) ; //给i节点解锁

}

static void write_inode (struct m_inode * inode) //写i节点

{

.....

lock_inode (inode) ; //给i节点加锁

if ( ! inode->i_dirt|| ! inode->i_dev) {

unlock_inode (inode) ;

return;

}

```

.....

```
bh->b_dirt=1;
```

```
inode->i_dirt=0;
```

```
brelse (bh) ;
```

```
unlock_inode (inode) ; //给i节点解锁
```

```
}
```

```
static inline void lock_inode (struct m_inode * inode)
```

```
{
```

```
cli () ;
```

```
while (inode->i_lock) //如果i节点已经加锁
```

```
sleep_on (&inode->i_wait) ; //将进程挂起
```

```
inode->i_lock=1; //给i节点加锁
```

```
sti () ;
```

```
}
```

```
static inline void unlock_inode (struct m_inode * inode)
```

```
{
```

```
inode->i_lock=0; //给i节点解锁
```

```
wake_up (&inode->i_wait) ; //唤醒等待i节点解锁的进程  
  
}
```

super_block[8]中s_lock和*s_wait联用时的代码如下:

//代码路径: fs/inode.c:

```
static struct super_block * read_super (int dev) //读超级块  
{  
  
.....  
  
s->s_time=0;  
  
s->s_rd_only=0;  
  
s->s_dirt=0;  
  
lock_super (s) ; //给超级块加锁  
  
if ( ! (bh=bread (dev, 1) ) ) {  
  
s->s_dev=0;  
  
free_super (s) ;
```



```

return NULL;

}

.....

s->s_imap[0]->b_data[0]=1;

s->s_zmap[0]->b_data[0]=1;

free_super (s) ; //给超级块解锁

return s;

}

void put_super (int dev) //释放超级块

{

.....

if (sb->s_imount) {

printf ("Mounted disk changed-tssk,tssk\n\r") ;

return;

}

lock_super (s) ; //给超级块加锁

sb->s_dev=0;

```

```

for (i=0; i<I_MAP_SLOTS; i++)

brelse (sb->s_imap[i]) ;

for (i=0; i<Z_MAP_SLOTS; i++)

brelse (sb->s_zmap[i]) ;

free_super (sb) ;

free_super (s) ; //给超级块解锁

.....

}

static void lock_super (struct super_block * sb)

{

cli () ;

while (sb->s_lock) //如果超级块已经加锁

sleep_on (& (sb->s_wait) ) ; //将进程挂起

sb->s_lock=1; //给超级块加锁

sti () ;

}

static void free_super (struct super_block * sb)

```

```
{  
  
cli ( ) ;  
  
sb->s_lock=0; //给超级块解锁  
  
wake_up ( & (sb->s_wait) ) ; //唤醒等待超级块解锁的进程  
  
sti ( ) ;  
  
}
```

7.5.5 补充request的作用

缓冲块、i节点、超级块等结构中设置的字段，为进程共享缓冲块建立了基础，解决了“能共享还是不能共享”的问题。下面介绍如何更高效地共享缓冲块。

本节到这里介绍了缓冲块在进程方向上延伸的使用问题。下面介绍在硬盘方向上延伸的使用问题。我们来看请求项的数据结构，代码如下：

//代码路径：kernel/blk_drv/Blk.h:

```
struct request{

    int dev; /*-1 if no request*/

    int cmd; /*READ or WRITE*/

    int errors;
```

```
unsigned long sector;

unsigned long nr_sectors;

char * buffer;

struct task_struct * waiting;

struct buffer_head * bh;

struct request * next;

}
```

请求项request要和硬盘进行交互，所以要明确是读交互还是写交互，为此设计了cmd字段；此外，还需要明确是哪个缓冲块要进行交互，比如*bh和*buffer字段；还需要考虑数据块与扇区的映射规则，比如sector和nr_sectors字段；还需要考虑如果交互出现了问题怎么办，用errors记录出现问题的次数。这些字段都是为了与硬盘交互设置的。

硬盘方向完全是某个缓冲块和某个数据块一对一的交互，不存在共享问题，所以请求项中也就没有类似**b_count**的字段，请求项对交互状况的记录，只存在两种状态：忙、空闲。因此**dev**字段不仅代表设备号，还可以通过它来判断请求项是否正在被占用，代码如下：

//代码路径：kernel/blk_drv/ll_rw_block.c:

```
void blk_dev_init (void)

{

int i;

for (i=0; i<NR_REQUEST; i++) {

request[i].dev=-1; //把设备号设置为-1，标志着请求项都是空闲的

request[i].next=NULL;

}

}
```

```

static void make_request (int major,int rw,struct buffer_head * bh)

{

.....

/*find an empty request*/

while (--req>=request) //找一个空闲的请求项

if (req->dev<0) //小于0，那肯定就是-1了，说明空闲

break;

.....

req->dev=bh->b_dev; //用设备号来设置dev，那就肯定不是-1
了，设备号没有-1这个值

req->cmd=rw;

req->errors=0;

req->sector=bh->b_blocknr<<1;

req->nr_sectors=2;

req->buffer=bh->b_data;

req->waiting=NULL;

req->bh=bh;

```

```
req->next=NULL;
```

```
add_request (major+blk_dev,req) ;
```

```
}
```

```
//代码路径: kernel/blk_drv/Blk.h:
```

```
extern inline void end_request (int uptodate)
```

```
{
```

```
.....
```

```
wake_up (&CURRENT->waiting) ;
```

```
wake_up (&wait_for_request) ;
```

```
    CURRENT->dev=-1; //一个请求项的任务完成后，立即将这个请求项设置为空闲
```

```
    CURRENT=CURRENT->next;
```

```
}
```

另外值得注意的是，请求项request设置为32，尽可能地实现了主机和硬盘数据交互的平衡，但这种平衡并不绝对，比如说，写盘过于频

繁，或者由于硬盘自身出现故障，导致数据交互失败，就有可能在请求项中积压数据，最终导致请求项不够用了。那么内核即便为进程申请到了缓冲块，而由于没有请求项，进程也只能被挂起，同样也需要字段来记录哪个进程被挂起了。
*waiting记录挂起的进程，代码如下：

//代码路径: kernel/blk_drv/ll_rw_block.c:

```
static void make_request (int major,int rw,struct buffer_head * bh)
{
    .....

    if (req < request) { //没找到空闲的请求项

        if (rw_ahead) {

            unlock_buffer (bh) ;

            return;

        }
    }
}
```

```
sleep_on (&wait_for_request) ; //进程就挂起了

goto repeat;

}

}
```

等到有了空闲请求项，再唤醒进程，代码如下：

```
//代码路径： kernel/blk_drv/blk.h:

extern inline void end_request (int uptodate)

{

.....

wake_up (&CURRENT->waiting) ;

wake_up (&wait_for_request) ; //等待请求项的进程被唤醒

CURRENT->dev=-1;

CURRENT=CURRENT->next;

}
```

同理，多个进程也可能都由于等待某个请求项被挂起，`*waiting`一个字段是无法记录的，同样需要进程等待队列解决这个问题。本小节前面也提到了用`*b_wait`记录等待缓冲块解锁的进程，这里的`*waiting`与此类似，都需要用到进程等待队列的技巧来完成对多个等待进程的记录。

7.6 实例1：关于缓冲块的进程等待队列

下面我们通过一个多进程操作相同文件的案例，一方面把共享地问题形象地进行体现，另一方面把进程等待队列的原理讲解清楚。

假设硬盘上已有一个文件名为hello.txt的文件，这个文件的大小为700 B，小于一个数据块大小（1 KB），被载入缓冲区后，一个缓冲块就可以承载其全部内容，这三个进程一旦开始操作这个文件，就相当于在依托系统操作同一个缓冲块，这样就会产生进程等待队列。本节将详细介绍该队列的产生过程，以及队列中进程的唤醒过程。

下面我们来介绍实例1的场景。

进程A是一个读盘进程，目的是将hello.txt文件中的100字节读入buffer[100]，代码如下：

```
void FunA () ;
```

```
void main ()
```

```
{
```

```
.....
```

```
FunA () ;
```

```
.....
```

```
}
```

```
void FunA ()
```

```
{
```

```
char buffer[100];
```

```
int i,j;
```

```
//打开文件
```

```
int fd=open ("/mnt/user/user1/user2/hello.txt", O_RDWR,  
0644) ) ;
```

```
//读文件
```

```
read (fd,buffer,sizeof (buffer) ) ;
```

```
//关闭文件
```

```
close (fd) ;
```

```
for (i=0; i<1000000; i++) //消耗时间片
```

```
{
```

```
for (j=0; j<1000000; j++)
```

```
{
```

```
;
```

```
}
```

```
}
```

```
return;
```

```
}
```

进程B也是一个读盘进程，目的是将hello.txt文件中的200字节读入buffer[200]，代码如下：

```
void FunB () ;

void main ()

{

.....

FunB () ;

.....

}

void FunB ()

{

char buffer[200];

int i,j;

//打开文件

int fd=open ("/mnt/user/user1/user2/hello.txt", O_RDWR,
0644) ) ;
```

```
//读文件

read (fd,buffer,sizeof (buffer) ) ;

//关闭文件

close (fd) ;

for (i=0; i<1000000; i++) //消耗时间片

{

for (j=0; j<1000000; j++)

{

;

}

}

return;

}
```

进程C是一个写盘进程，目的是往hello.txt文件中写入str1[]中的字符“ABCDE”，代码如下：

```
void FunC () ;
```

```
void main ()
```

```
{
```

```
.....
```

```
FunC () ;
```

```
.....
```

```
}
```

```
void FunC ()
```

```
{
```

```
char str1[]="ABCDE";
```

```
int i,j;
```

```
//打开文件
```

```
int fd=open ("/mnt/user/user1/user2/hello.txt", O_RDWR,  
0644) ) ;
```

```
//写文件
```

```
write (fd,str1, strlen (str1) ) ;
```

```
//关闭文件
```

```
close (fd) ;

for (i=0; i<1000000; i++) //消耗时间片

{

for (j=0; j<1000000; j++)

{

;

}

}

return;

}
```

这三个进程执行顺序为：进程A先执行，之后进程B执行，最后进程C执行。这三个进程没有父子关系。

下面我们来看具体的执行过程。

1.进程A读取文件后被挂起

进程A启动后，执行“int fd=open
（“/mnt/user/user1/user2/hello.txt”， O_RDWR,
0644） ）；”，open（）函数最终会映射到
sys_open（）函数去执行。sys_open（）函数的执
行情况，已在第5章中介绍。代码如下：

//代码路径：fs/open.c:

```
int sys_open（const char * filename,int flag,int mode）  
  
{  
  
.....  
  
    她（current-> filp[fd]=f）-> f_count++; //将进程A的*filp[20]与  
file_table[64]对应项挂接，并增加文件句柄计数  
  
.....  
  
    if（（i=open_namei（filename,flag,mode， &inode））<0）{//获  
取hello.txt文件i节点  
  
.....
```

```
f->f_mode=inode->i_mode; //用该i节点属性，设置文件属性

f->f_flags=flag; //用flag参数，设置文件操作方式

f->f_count=1; //将文件引用计数加1

f->f_inode=inode; //文件与i节点建立关系

f->f_pos=0; //将文件读写指针设置为0

return (fd); //把文件句柄返给用户空间

}
```

执行情景如图7-12所示。

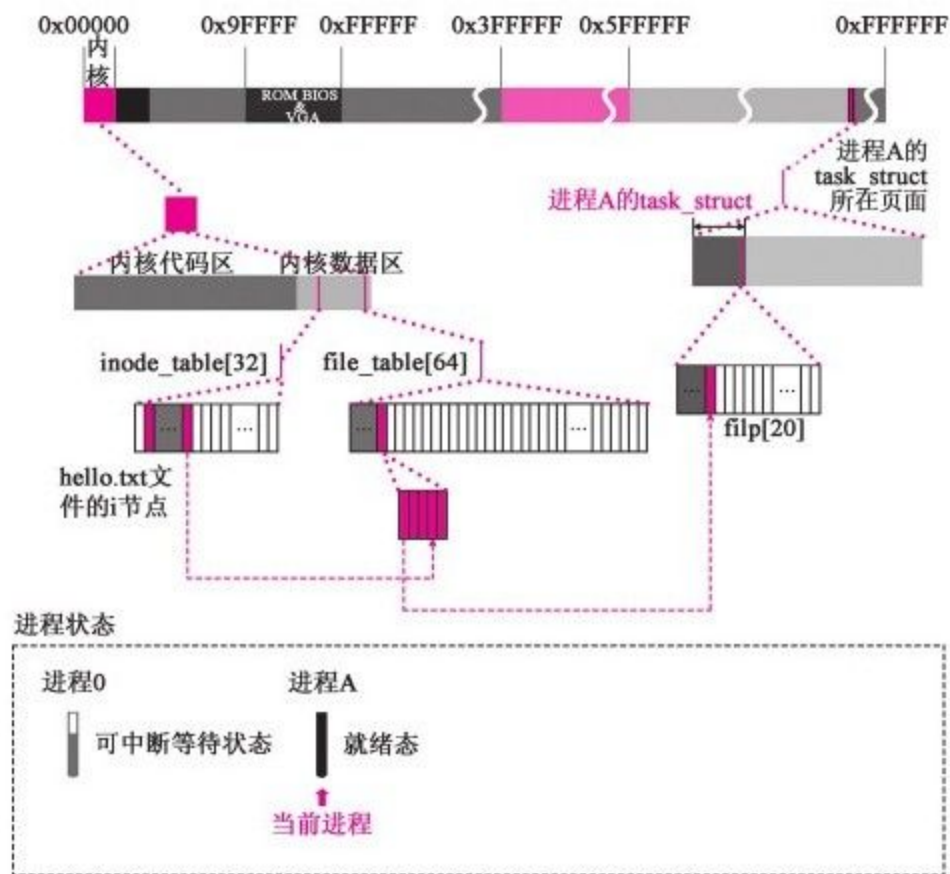


图 7-12 系统为进程A打开hello.txt文件

之后执行“read (fd,buffer,sizeof (buffer)) ”， read () 函数最终会映射到 sys_read () 函数去执行；之后， sys_read () 函数调用file_read () 函数来读取文件内容；

file_read () 函数调用bread () 函数从硬盘上读取数据。执行代码如下：

```
//代码路径: fs/read_write.c:
```

```
int sys_read (unsigned int fd,char * buf,int count) //从hello.txt文件中读数据
```

```
{//fd是文件句柄, buf是用户空间指针, count是要读取的字节数
```

```
.....
```

```
if (S_ISDIR (inode->i_mode) ||S_ISREG (inode->i_mode) ) {
```

```
if (count+file->f_pos>inode->i_size)
```

```
count=inode->i_size-file->f_pos;
```

```
if (count<=0)
```

```
return 0;
```

```
return file_read (inode,file,buf,count) ; //读取进程指定数据
```

```
}
```

```
printk (" (Read) inode->i_mode=%06o\n\r", inode->i_mode) ;
```

```
return-EINVAL;
```

```

    }

//代码路径: fs/file_dev.c:

int file_read (struct m_inode * inode,struct file * filp,char * buf,int
count)

{

.....

if (nr=bmap (inode, (filp->f_pos) /BLOCK_SIZE) ) {

if ( ! (bh=bread (inode->i_dev,nr) ) ) //从硬盘上读取数据

break;

}else

bh=NULL;

.....

}

```

进入**bread** () 函数后的执行过程，我们在3.3.1节中已经详细说明。执行代码如下：

//代码路径: fs/buffer.c:

```
struct buffer_head * bread (int dev,int block) //从硬盘上读取数据
```

```
{
```

```
.....
```

```
if (! (bh=getblk (dev,block) )) //申请一个空闲的缓冲块
```

```
.....
```

```
ll_rw_block (READ,bh) ; //将该缓冲块加锁并与请求项绑定,  
发送读盘指令
```

```
wait_on_buffer (bh) ; //如果有等待缓冲块解锁的进程, 就将其  
挂起
```

```
if (bh->b_uptodate)
```

```
return bh; .....
```

```
}
```

进程A的挂起工作是在wait_on_buffer () 函数中完成的, 执行代码如下:

//代码路径: fs/buffer.c:


```
static inline void wait_on_buffer (struct buffer_head * bh) //如果有  
等待缓冲块解锁的进程，就将其挂起
```

```
{  
  
cli () ; //关中断  
  
while (bh->b_lock) //检测缓冲块是否已经加锁  
  
sleep_on (&bh->b_wait) ; //将等待该缓冲块的进程 (进程A)  
挂起，并切换进程  
  
sti () ; //开中断  
  
}
```

在ll_rw_block () 函数执行时，缓冲块已经被加锁（本书3.3.1.3节中已介绍）。while (bh->b_lock) 条件为真，之后调用sleep_on () 函数，传递的实参是&bh->b_wait。其中bh->b_wait表示等待该缓冲块解锁的进程指针。由于系统在初始化时，已经将所有缓冲块中b_wait的值设置为NULL，此缓冲块又是新申请的，从来没有被其

他进程用过，所以此时bh->b_wait的值为NULL。下面进入sleep_on（）函数，执行代码如下：

```
//代码路径： kernel/sched.c:

void sleep_on (struct task_struct ** p)

{

    struct task_struct * tmp;

    if (! p)

        return;

    if (current==& (init_task.task) )

        panic ("task[0]trying to sleep") ;

    tmp=*p; //此时tmp中保存的是NULL

    *p=current; //p中保存的是进程A的指针

    current->state=TASK_UNINTERRUPTIBLE; //将进程A设置为不可中断等待状态

    schedule（）； //切换进程
```

```
if (tmp)

tmp->state=0;

}
```

从前面对sleep_on（）函数的实参的介绍中我们得知，*p指向的是bh->b_wait。*p中保存了进程A的指针，意味着进程A此时正在等待bh这个缓冲块解锁。

进程A被挂起后，调用schedule（）函数，切换到进程B执行。

与此同时，硬盘也正在向数据寄存器端口中传递数据，此情景如图7-13所示。



图 7-13 系统为进程A读取hello.txt的数据并将进程A挂起

图7-13中代表进程A的进程条已经变成了灰色，表示进程A已经挂起。

值得注意的是，代码中的tmp存储在进程A的内核栈中，存储的是NULL,bh->b_wait此时存储的是进程A的指针，如图7-14所示。

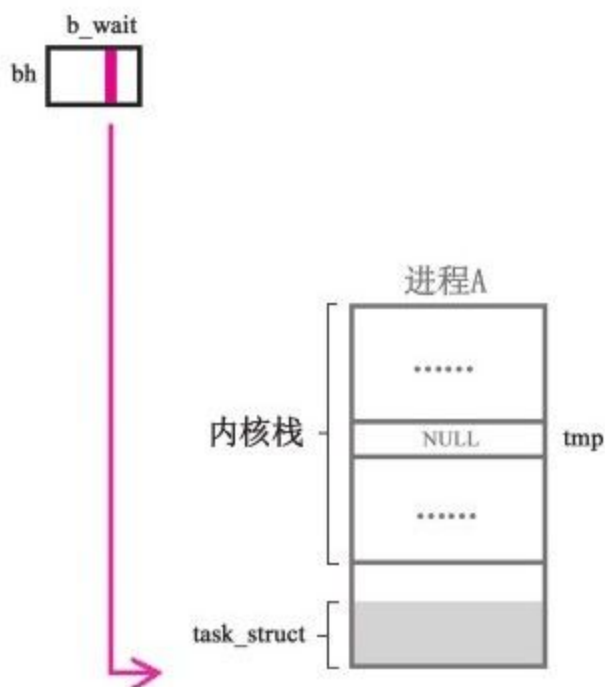


图 7-14 进程A被挂起

2.进程B读取文件后被挂起

进程B首先执行“int fd=open
 (“/mnt/user/user1/user2/hello.txt”, O_RDWR,
 0644)) ; ”这行代码。open () 函数最终会映射
 到sys_open () 函数去执行； sys_open () 函数会
 在文件管理表file_table[64]中新申请一个空闲表

项，让进程B task_struct中的*filp[20]与file_table[64]空表项挂接。虽然进程B和进程A打开的是相同的文件，但实例1中，这两个进程彼此对文件的操作没有关系，所以需要两套账本。

执行代码如下：

//代码路径： fs/open.c:

```
int sys_open (const char * filename,int flag,int mode)
```

```
{
```

```
.....
```

```
for (fd=0; fd<NR_OPEN; fd++)
```

```
if (! current->filp[fd])
```

```
break;
```

```
.....
```

```
for (i=0; i<NR_FILE; i++, f++)
```

```
if (! f->f_count) break;
```

```

.....

    (current->filp[fd]=f) -> f_count++; //将进程B的*filp[20]与
file_table[64]对应项挂接，并增加文件句柄计数

    if ( (i=open_namei (filename,flag,mode, &inode) ) < 0) { //获
取hello.txt文件i节点

.....

f->f_mode=inode->i_mode; //用该i节点属性，设置文件属性

f->f_flags=flag; //用flag参数，设置文件操作方式

f->f_count=1; //将文件引用计数加1

f->f_inode=inode; //文件与i节点建立关系

f->f_pos=0; //将文件读写指针设置为0

return (fd) ; //把文件句柄返给用户空间

}

```

挂接的情景如图7-15所示。

硬盘还在不断地读出数据，刚才进程A的读盘请求还没有完成。

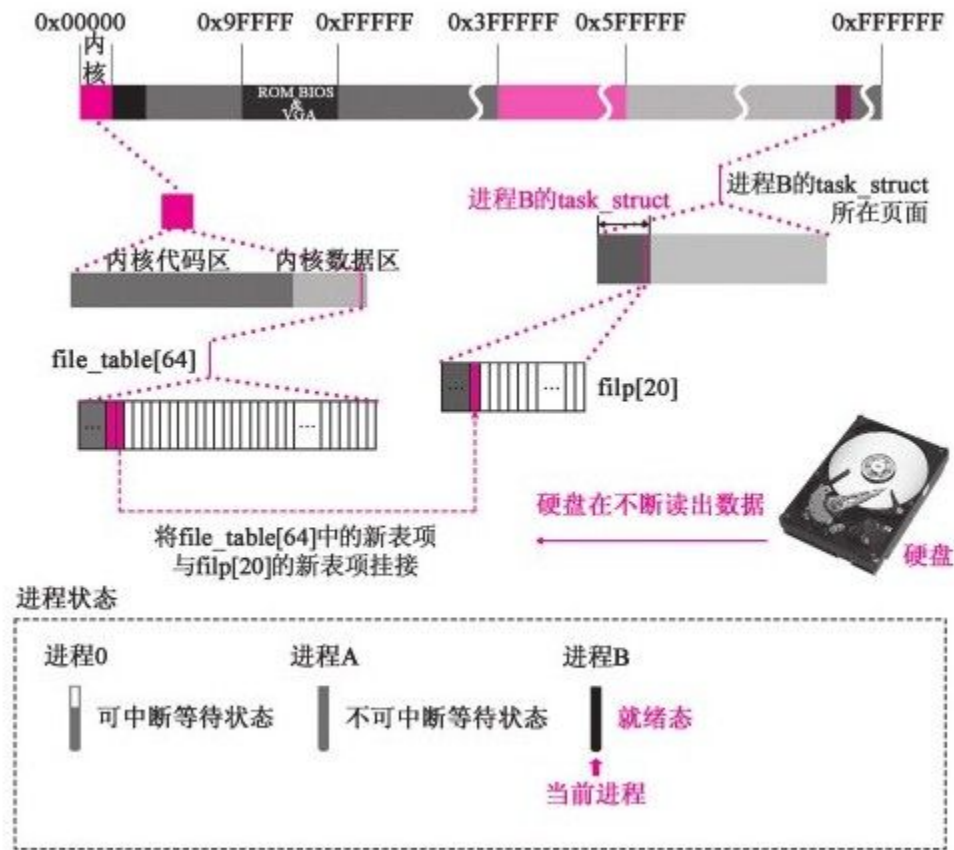


图 7-15 系统将filp[20]与file_table[64]挂接

之后调用open_namei () 函数，获取hello.txt文件的i节点，并最终将i节点与file_table[64]相挂接，执行代码如下：

//代码路径： fs/open.c:

```
int sys_open (const char * filename,int flag,int mode)
```



```
{  
  
.....  
  
    if ( (i=open_namei (filename,flag,mode, &inode) ) < 0) { //获取hello.txt文件i节点  
  
        .....  
  
        f->f_mode=inode->i_mode; //用该i节点属性, 设置文件属性  
  
        f->f_flags=flag; //用flag参数, 设置文件操作方式  
  
        f->f_count=1; //将文件引用计数加1  
  
        f->f_inode=inode; //文件与i节点建立关系  
  
        f->f_pos=0; //将文件读写指针设置为0  
  
        return (fd) ; //把文件句柄返给用户空间  
  
    }
```

值得注意的是, 此次获取hello.txt文件的i节点, 与进程A获取该i节点有所不同, 代码如下:

//代码路径: fs/namei.c:

```

int open_namei (const char * pathname,int flag,int mode,

struct m_inode ** res_inode)

{

.....

if (flag&O_EXCL)

return-EEXIST;

if ( ! (inode=iget (dev,inr) ) ) //获取i节点

return-EACCES;

.....

}

```

//代码路径: fs/namei.c:

```

struct m_inode * iget (int dev,int nr)

{

.....

```

项 empty=get_empty_inode () ; //在inode_table[32]中申请空闲的表项

```

.....

```

```
while (inode < NR_INODE + inode_table) { //遍历整个
inode_table[32]
```

```
if (inode->i_dev != dev || inode->i_num != nr) { //如果没有找到现
成的表项，就继续找
```

```
.....
```

```
continue;
```

```
}
```

```
wait_on_inode (inode) ;
```

```
if (inode->i_dev != dev || inode->i_num != nr) {
```

```
.....
```

```
continue;
```

```
}
```

```
inode->i_count++; //找到了现成的hello.txt文件的i节点，引用计
数增加
```

```
.....
```

```
if (empty) //在inode_table[32]中找到的空闲表项已经没用了，将
其释放
```

```
iput (empty) ;
```

```
return inode; //将hello.txt文件的i节点返回
```

}

.....

}

申请空闲i节点的情景如图7-16所示。

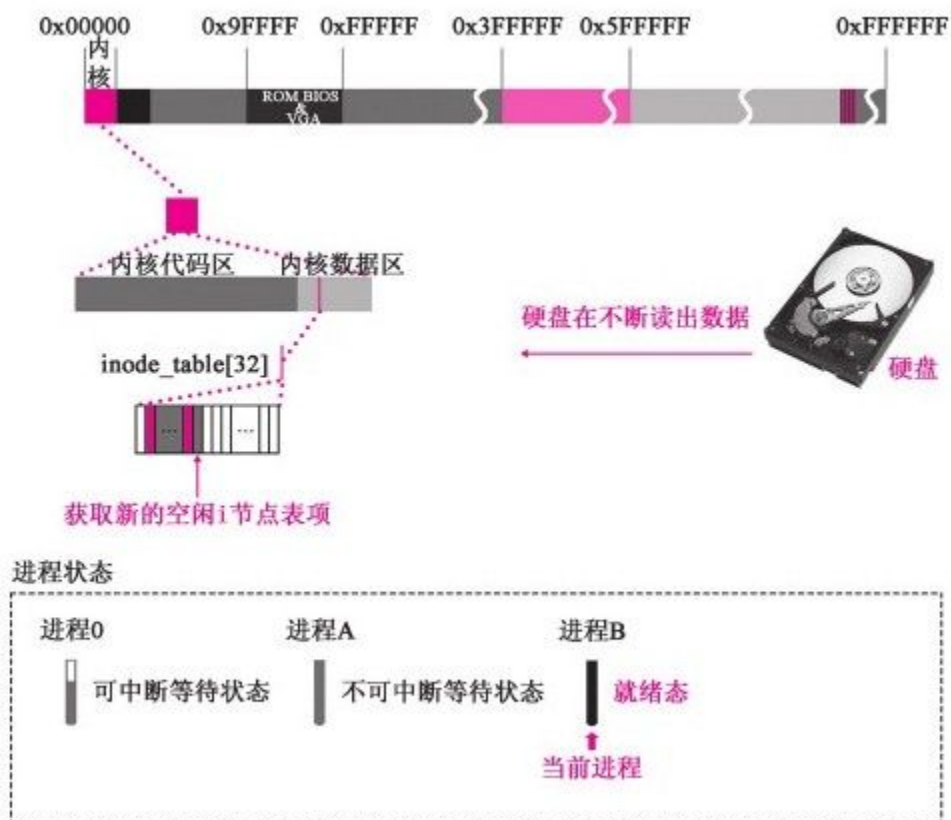


图 7-16 系统为载入i节点创造条件

一个文件只能对应一个i节点。进程A和进程B对文件的操作，需要两本账来记录。但它们操作的hello.txt文件的i节点只能有一个，而进程A已经把i节点载入了inode_table[32]，现在进程B就要沿用这个i节点。

上述代码中对i节点的操作情景如图7-17所示。

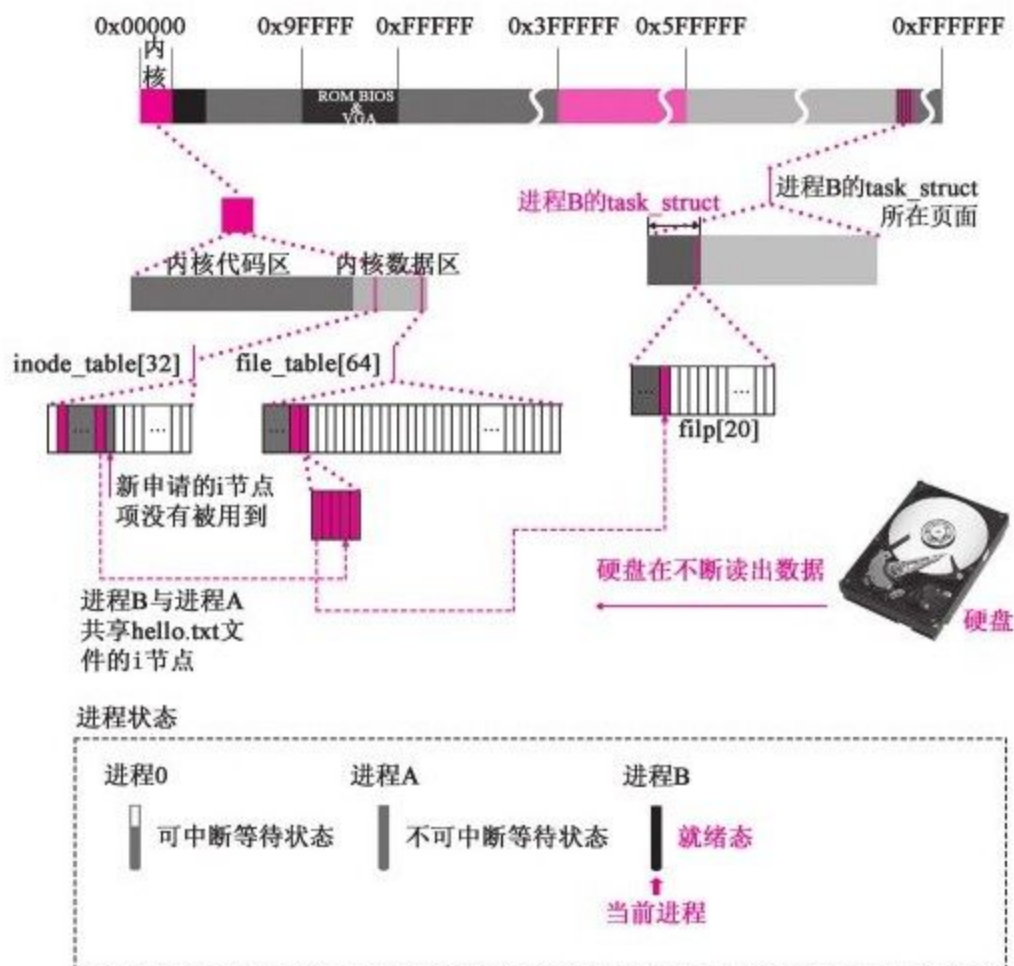


图 7-17 系统为进程B找到了hello.txt文件已经载入的i节点

文件打开后，进程B执行“read (fd,buffer,sizeof (buffer)) ; ”这行代码，读取hello.txt文件的内容。

read () 函数最终会映射到sys_read () 函数去执行，之后，sys_read () 函数调用file_read () 函数来读取文件内容，file_read () 函数调用bread () 函数从硬盘上读取数据，执行代码如下：

```
//代码路径: fs/read_write.c:
```

```
int sys_read (unsigned int fd,char * buf,int count) //从hello.txt文件中读数据
```

```
{//fd是文件句柄， buf是用户空间指针， count是要读取的字节数
```

```
.....
```

```
if (S_ISDIR (inode->i_mode) ||S_ISREG (inode->i_mode) ) {
```

```
if (count+file->f_pos>inode->i_size)
```

```
count=inode->i_size-file->f_pos;
```

```
if (count<=0)
```

```
return 0;
```

```
return file_read (inode,file,buf,count) ; //读取进程指定数据
```

```

    }

    printk (" (Read) inode->i_mode=%06o\n\r", inode->
i_mode) ;

    return-EINVAL;

}

//代码路径: fs/file_dev.c:

int file_read (struct m_inode * inode,struct file * filp,char * buf,int
count)

{

.....

if (nr=bmap (inode, (filp->f_pos) /BLOCK_SIZE) ) {

if (! (bh=bread (inode->i_dev,nr) ) ) //从硬盘上读取数据

break;

}else

bh=NULL;

.....

}

```

进入bread（）函数后的执行过程，我们在3.3.1节中已经详细说明。执行代码如下：

```
//代码路径： fs/buffer.c:

struct buffer_head * bread（int dev,int block） //从硬盘上读取数据
{
.....

if（!（bh=getblk（dev,block））） //申请一个空闲的缓冲块

.....

ll_rw_block（READ,bh）； //将该缓冲块加锁并与请求项绑定，
发送读盘指令

wait_on_buffer（bh）； //如果有等待缓冲块解锁的进程，就将其
挂起

if（bh->b_uptodate）

return bh;

.....

}
```

其中getblk（）函数和ll_rw_block（）函数的执行情景有所不同。进入getblk（）函数后，由于hello.txt文件对应的数据块已被载入缓冲区，直接返回，代码如下：

```
//代码路径： fs/buffer.c:

struct buffer_head * getblk（int dev,int block） //申请缓冲块

{

.....

    if（bh=get_hash_table（dev,block）） //此时在哈希表中可以找到指定的缓冲块

return bh; //直接返回bh指针

.....

}
```

之后执行ll_rw_block（）函数。由于缓冲块已经被加锁，所以进程B将因等待该缓冲块解锁

而被系统挂起，执行代码如下：

```
//代码路径: kernel/blk_drv/ll_rw_block.c:

void ll_rw_block (int rw,struct buffer_head * bh)

{

unsigned int major;

if ( (major=MAJOR (bh->b_dev) ) >=NR_BLK_DEV||

! (blk_dev[major].request_fn) ) {

printk ("Trying to read nonexistent block-device\n\r") ;

return;

}

make_request (major,rw,bh) ; //设置请求项

}

static void make_request (int major,int rw,struct buffer_head * bh)

{

.....

lock_buffer (bh) ; //将bh指向的缓冲块加锁
```

```
    if ( (rw==WRITE&&! bh->b_dirt) || (rw==READ&&bh->
b_uptodate) ) {
```

```
        unlock_buffer (bh) ;
```

```
        return;
```

```
    }
```

```
    .....
```

```
}
```

```
static inline void lock_buffer (struct buffer_head * bh) //给缓冲块加
锁
```

```
{
```

```
    cli () ;
```

```
    while (bh->b_lock) //如果缓冲块已经加锁
```

```
        sleep_on (&bh->b_wait) ; //就将等待缓冲块解锁的进程挂起
```

```
    bh->b_lock=1; //程序执行到这里，说明缓冲块此时没有加锁，
于是给它加锁
```

```
    sti () ;
```

```
}
```

接下来执行sleep_on () 函数。因为实例1中进程A和进程B操作的是同一文件，对应相同的缓冲块bh，该缓冲块中b_wait的值被设置为进程A的task_struct指针，所以此次执行sleep_on () 函数的情景，与前面进程A执行时的情景完全不同，代码如下：

//代码路径： kernel/sched.c:

```
void sleep_on (struct task_struct ** p)
```

```
{
```

```
struct task_struct * tmp;
```

```
if (! p)
```

```
return;
```

```
if (current==& (init_task.task) )
```

```
panic ("task[0]trying to sleep") ;
```

```
tmp=*p; //此时tmp中保存的是进程A的task_struct指针
```

```
*p=current; /*p中保存的是进程B的task_struct指针  
  
current->state=TASK_UNINTERRUPTIBLE; //将进程B设置为不可中断等待状态  
  
schedule (); //切换进程  
  
if (tmp)  
  
tmp->state=0;  
  
}
```

进程B被挂起后，调用schedule（）函数，切换到进程C执行。

与此同时，硬盘也正在向数据寄存器端口中传递数据，此情景如图7-18所示。



图 7-18 进程B被挂起的情景

值得注意的是，代码中的tmp存储在进程B的内核栈中，存储的是进程A的task_struct指针，bh->b_wait此时存储的是进程B的指针，如图7-19所示。

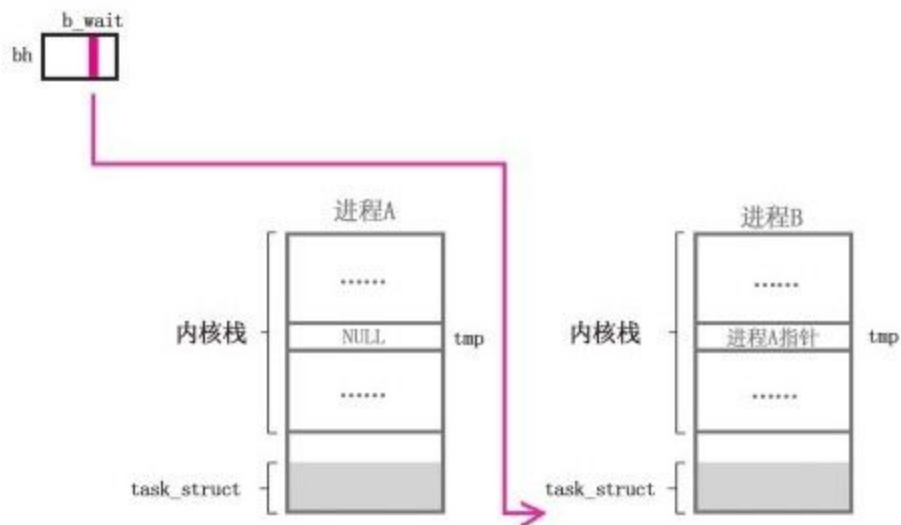


图 7-19 进程B被挂起，构成进程等待队列的情景

3.进程C写文件后被挂起

进程C开始执行后，同样操作hello.txt文件，向该文件中写入数据。进程C执行的技术路线与进程B大体一致，先执行“int fd=open

（“/mnt/user/user1/user2/hello.txt”， O_RDWR， 0644））；”这行代码，open（）函数最终会映射

到sys_open () 函数去执行。sys_open () 函数最终的执行结果为，在file_table[64]中再次申请一个空闲表项，进程C task_struct中的*filp[20]与file_table[64]中的空闲表项挂接。

之后sys_open () 函数调用open_namei () 函数，同样会在i节点表inode_table[32]中找到该文件的i节点，该i节点的引用计数再次加1，执行代码如下：

```
//代码路径： fs/open.c:

int sys_open (const char * filename,int flag,int mode)

{

.....

    (current-> filp[fd]=f) -> f_count++; //将进程C的*filp[20]与
file_table[64]对应项挂接， 并增加文件句柄计数

.....
```

```
if ( (i=open_namei (filename,flag,mode, &inode) ) < 0) { //获  
取hello.txt文件i节点
```

```
.....
```

```
f->f_mode=inode->i_mode; //用该i节点属性，设置文件属性
```

```
f->f_flags=flag; //用flag参数，设置文件操作方式
```

```
f->f_count=1; //将文件引用计数加1
```

```
f->f_inode=inode; //文件与i节点建立关系
```

```
f->f_pos=0; //将文件读写指针设置为0
```

```
return (fd) ; //把文件句柄返给用户空间
```

```
}
```

此情景如图7-20所示。

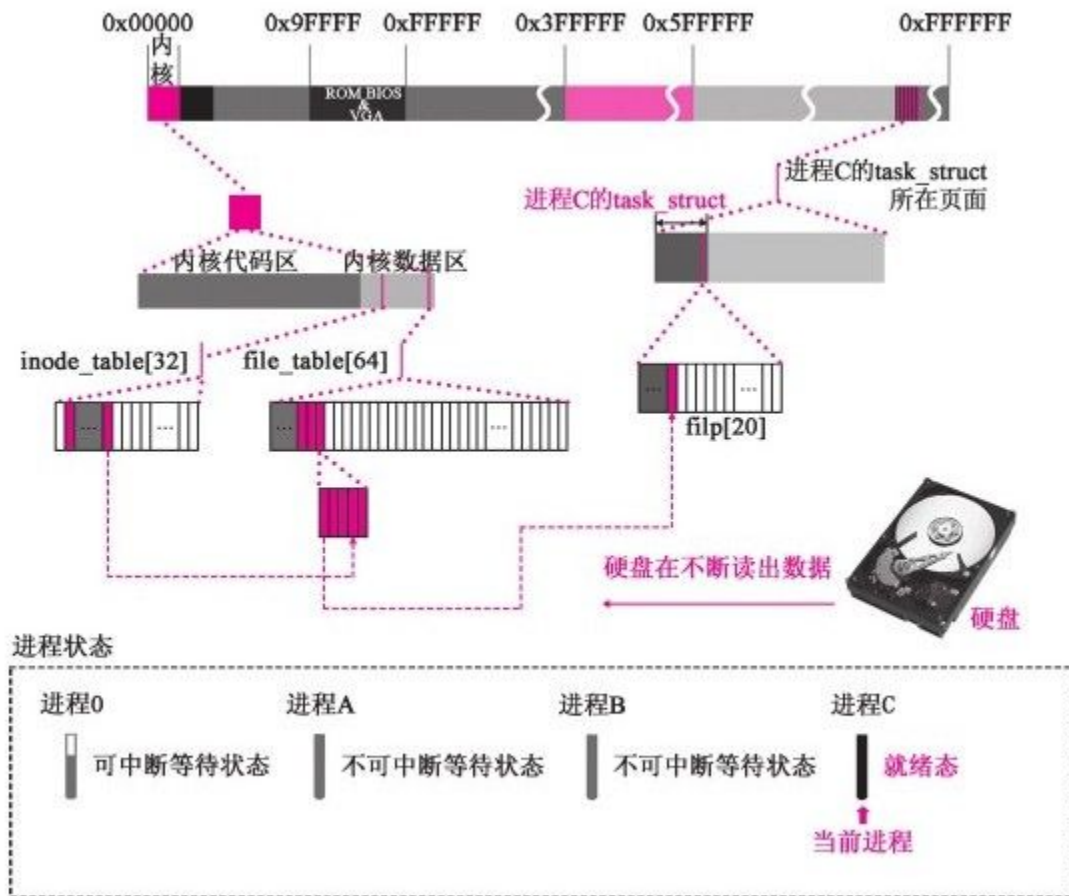


图 7-20 进程C准备读取hello.txt文件

进程C继续执行“write (fd, str1, strlen (str1)) ; ”这行代码，往hello.txt文件写入数据。

`write ()` 函数最终会映射到`sys_write ()` 函数去执行, `sys_write ()` 函数调用`file_write ()` 函数来读取文件内容, `file_write ()` 函数调用`bread ()` 函数从硬盘上读取数据, 执行代码如下:

//代码路径: fs/read_write.c:

```
int sys_write (unsigned int fd,char * buf,int count) //向hello.txt文件中写入数据
```

```
{//fd是文件句柄, buf是用户空间指针, count是要写入的字节数
```

```
.....
```

```
if (S_ISBLK (inode->i_mode) )
```

```
    return block_write (inode->i_zone[0], &file->f_pos,buf,count) ;
```

```
if (S_ISREG (inode->i_mode) )
```

```
    return file_write (inode,file,buf,count) ; //写入进程指定数据
```

```
    printk (" (Write) inode->i_mode=%06o\n\r", inode->i_mode) ;
```

```
return-EINVAL;
```

```
}
```

```
//代码路径: fs/file_dev.c:
```

```
int file_write (struct m_inode * inode,struct file * filp,char * buf,int  
count)
```

```
{
```

```
.....
```

```
if (! (block=create_block (inode,pos/BLOCK_SIZE) ) )
```

```
break;
```

```
if (! (bh=bread (inode->i_dev,nr) ) ) //往硬盘上写入数据
```

```
c=pos%BLOCK_SIZE;
```

```
p=c+bh->b_data;
```

```
bh->b_dirt=1;
```

```
.....
```

```
}
```

进入bread（）函数后的执行过程，与进程B一致，执行代码如下：

//代码路径： fs/buffer.c:

struct buffer_head * bread（int dev,int block） //从硬盘上读取数据

{

.....

if（!（bh=getblk（dev,block））） //申请一个空闲的缓冲块

.....

ll_rw_block（READ,bh）； //将该缓冲块加锁并与请求项绑定，
发送读盘指令

wait_on_buffer（bh）； //如果有等待缓冲块解锁的进程，就将其
挂起

if（bh->b_uptodate）

return bh;

.....

}

进入getblk（）函数后，由于hello.txt文件对应的数据块已被载入缓冲区，直接返回，代码如下：

```
//代码路径： fs/buffer.c:
```

```
struct buffer_head * getblk（int dev,int block） //申请缓冲块
```

```
{
```

```
struct buffer_head * tmp, *bh;
```

```
repeat:
```

```
    if（bh=get_hash_table（dev,block）） //此时在哈希表中可以找到指定的缓冲块
```

```
        return bh; //直接返回bh指针
```

```
    tmp=free_list;
```

```
    .....
```

```
}
```

执行ll_rw_block () 函数。由于缓冲块已经被加锁，所以进程C也将因等待该缓冲块解锁而被系统挂起（操作的是相同的缓冲块），执行代码如下：

//代码路径: kernel/blk_drv/ll_rw_block.c:

```
void ll_rw_block (int rw,struct buffer_head * bh)

{

unsigned int major;

if ( (major=MAJOR (bh->b_dev) ) >=NR_BLK_DEV||

! (blk_dev[major].request_fn) ) {

printf ("Trying to read nonexistent block-device\n\r") ;

return;

}

make_request (major,rw,bh) ; //设置请求项

}
```



```

static void make_request (int major,int rw,struct buffer_head * bh)

{

.....

lock_buffer (bh) ; //将bh指向的缓冲块加锁

    if ( (rw==WRITE&&! bh->b_dirt) || (rw==READ&&bh->
b_uptodate) ) {

        unlock_buffer (bh) ;

        return;

    }

.....

}

static inline void lock_buffer (struct buffer_head * bh) //给缓冲块加
锁

{

    cli () ;

    while (bh->b_lock) //如果缓冲块已经加锁

        sleep_on (&bh->b_wait) ; //就将等待缓冲块解锁的进程挂起

    bh->b_lock=1; //程序执行到这里，说明缓冲块此时没有加锁，
于是给它加锁

```

```
sti () ;  
  
}
```

接下来执行`sleep_on ()` 函数。因为实例1中进程A、进程B和进程C操作的是同一文件，对应相同的缓冲块bh，我们已经介绍到该缓冲块中`b_wait`的值被设置为进程B的`task_struct`指针，所以此次执行`sleep_on ()` 函数的情景，与前面进程B执行时的情景完全不同，代码如下：

```
//代码路径： kernel/sched.c:  
  
void sleep_on (struct task_struct ** p)  
{  
  
    struct task_struct * tmp;  
  
    if (! p)  
  
        return;  
  
    if (current==& (init_task.task) )
```

```
panic ("task[0]trying to sleep") ;

tmp=*p; //此时tmp中保存的是进程B的task_struct指针

*p=current; //*p中保存的是进程C的task_struct指针

current->state=TASK_UNINTERRUPTIBLE; //将进程C设置为不可中断等待状态

schedule () ; //切换进程

if (tmp)

tmp->state=0;

}
```

进程C被挂起后，调用schedule（）函数，此时系统中已经没有就绪的进程了，因此切换到进程0执行。

与此同时，硬盘也正在向数据寄存器端口中传递数据，此情景如图7-21所示。

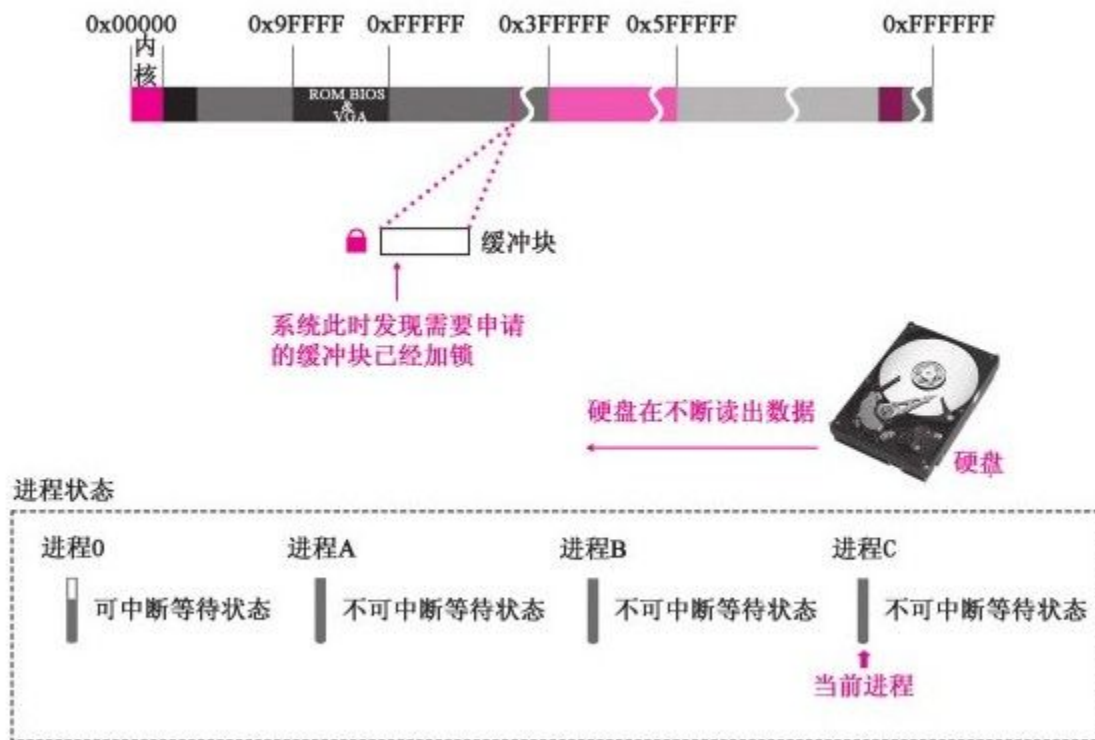


图 7-21 进程C被挂起后的情景

值得注意的是，代码中的tmp存储在进程C的内核栈中，存储的是进程B的task_struct指针，bh->b_wait此时存储的是进程C的指针，如图7-22所示。

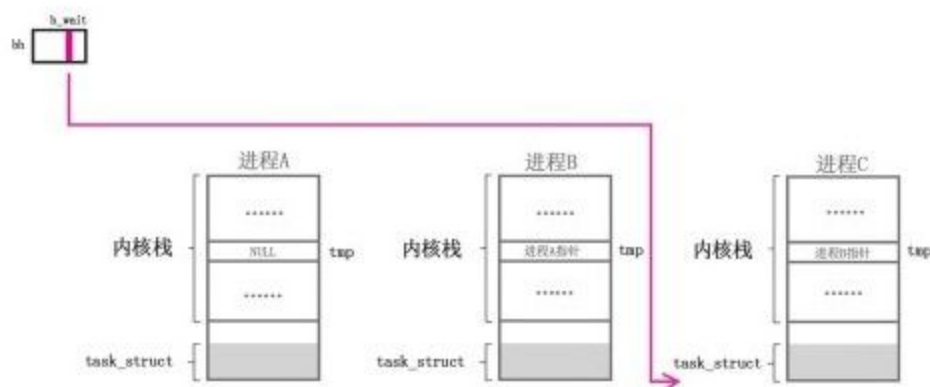


图 7-22 进程C被挂起，加入等待队列的情景

此时的情况是，三个进程因等待bh缓冲块解锁而被系统挂起，于是形成了一个等待队列。挂起前，每个进程的内核栈中都保存着前面被挂起进程的task_struct指针。图7-22表现的就是这个等待队列。这个队列的作用在于，等到缓冲块解锁时，操作系统可以根据每个被唤醒的进程中内核栈的记录，来唤醒在此进程挂起之前被挂起的进程，这样，所有等待缓冲块解锁的进程将被依次唤醒。具体过程将在下面详细介绍。

4.三个进程以相反的顺序被唤醒

此时进程A、进程B和进程C都已经被挂起了，系统中所有的进程又都处于非就绪态了。先以默认切换到进程0去执行，直到数据读取完毕，硬盘产生中断，如图7-23所示。



图 7-23 再次切换到进程0执行

硬盘中断产生后，中断服务程序将开始工作。此时硬盘已经将指定的数据全部载入缓冲块。中断服务程序开始工作后，将bh缓冲块解锁，并调用wake_up（）函数，将bh中wait字段所对应的进程（进程C）唤醒，执行代码如下：

```
//代码路径： kernel/blk_drv/Blk.h:
```

```
extern inline void end_request (int uptodate)
```

```
{
```

```
    DEVICE _OFF (CURRENT->dev) ;
```

```
    if (CURRENT->bh) {
```

```
        CURRENT->bh->b_uptodate=uptodate;
```

```
        unlock_buffer (CURRENT->bh) ; //将缓冲块解锁
```

```
    }
```

```
    .....
```

```
}
```

```
extern inline void unlock_buffer (struct buffer_head * bh)

{

if (! bh->b_lock)

printk (DEVICE_NAME": free buffer being unlocked\n") ;

bh->b_lock=0; //将缓冲块解锁

wake_up (&bh->b_wait) ; //唤醒等待缓冲块解锁的进程

}
```

调用wake_up () 函数时，传递的参数是&bh->b_wait。从进程等待队列图中不难发现，bh->b_wait指向的是进程C的task_struct指针，所以唤醒的是进程C。

wake_up () 函数执行代码如下：

//代码路径： kernel/sched.c:

```
void wake_up (struct task_struct ** p)
```



```

{

if (p && *p) {

    (**p).state=0; //这里将进程C设置为就绪态

*p=NULL;

}

}

```

此设置的情景如图7-24所示。

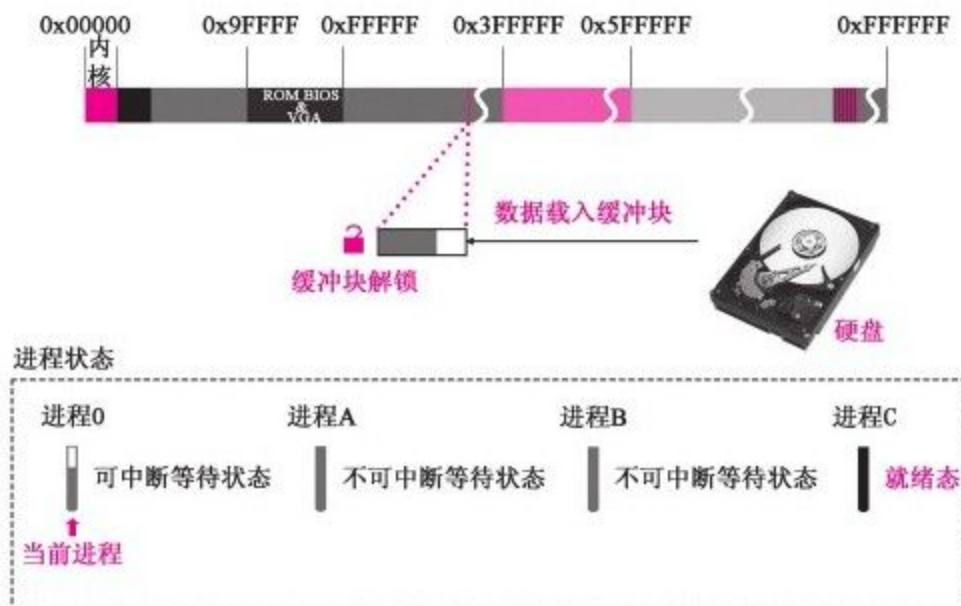


图 7-24 进程C被唤醒

中断服务程序结束后，再次返回进程0中，并切换到就绪的进程C。进程C是在sleep_on（）函数中，调用了schedule（）函数进行的进程切换，因此，进程C最终回到sleep_on（）函数中，首先要执行的代码如下：

```
//代码路径: kernel/sched.c:
```

```
void sleep_on (struct task_struct ** p)
```

```
{
```

```
.....
```

```
schedule（）；
```

```
if (tmp)
```

```
tmp->state=0; //将tmp所对应的进程设置为就绪态
```

```
}
```

我们来看图7-25。

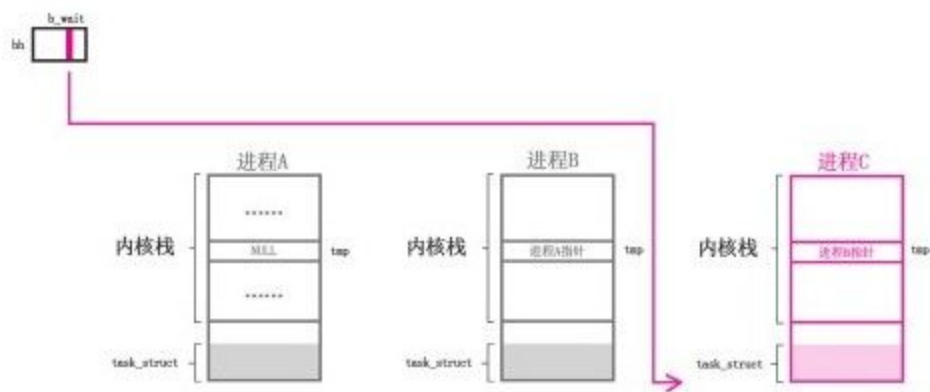


图 7-25 进程C被唤醒，退出进程等待队列

此时内核中程序在执行，所使用的是进程C的内核栈，这样tmp对应的是进程B的task_struct指针，所以此时是将进程B设置为就绪态。

此设置的情景如图7-26所示。



图 7-26 将进程B唤醒

内核继续执行，将把进程C的源程序中指定的str1这个字符数组中的数据写入hello.txt文件对应的缓冲块中，执行代码如下：

//代码路径: fs/file_dev.c:

```
int file_write (struct m_inode * inode,struct file * filp,char * buf,int  
count)
```

```
{
```

```
.....
```

```
if (pos > inode->i_size) {
```

```
inode->i_size=pos;
```

```
inode->i_dirt=1;
```

```
}
```

```
i+=c;
```

```
while (c-->0)
```

```

* (p++) =get_fs_byte (buf++) ; //将字符串写入缓冲块

brelse (bh) ;

.....

}

```

写入的情景如图7-27所示。

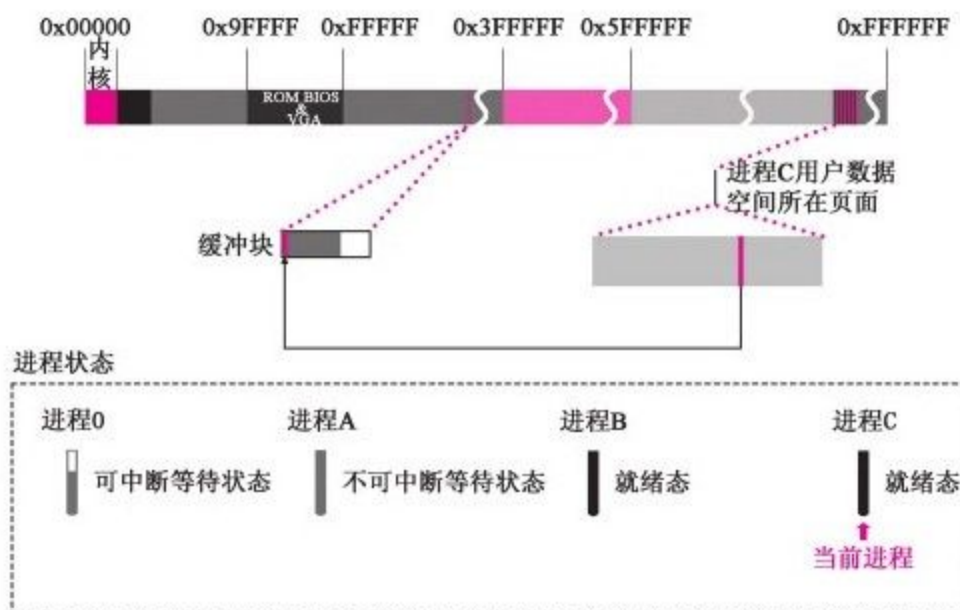


图 7-27 系统为进程C将数据写入指定缓冲块

之后返回进程C的用户程序中，执行如下代码：

```
for (i=0; i<1000000; i++) //消耗时间片

{

for (j=0; j<1000000; j++)

{

;

}

}
```

执行过程中，时钟中断不断产生，进程C的时间片被不断地削减，如图7-28所示。



图 7-28 进程C执行过程中，时间片不断削减

注意各个进程执行状态的进程条中，进程C的时间片在不断地削减，如图7-29所示。



图 7-29 进程C的时间片减少到0

进程C的时间片削减为0时，要切换进程了。前面已经介绍到，进程C被唤醒后，系统的第一件事就是将进程B设置为就绪态。此时系统中只有进程B和进程C两个进程是就绪态，进程C的时间片用完了，就会切换到进程B去执行，如图7-30所示。

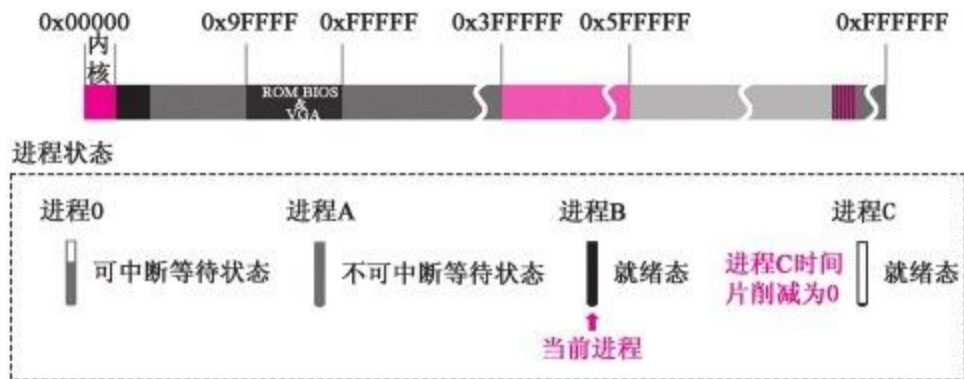


图 7-30 切换到进程B执行

进程B也是在sleep_on（）函数中，调用了schedule（）函数进行的进程切换，因此，进程B最终回到sleep_on（）函数中，首先要执行的代码如下：

```
//代码路径： kernel/sched.c:

void sleep_on（struct task_struct ** p)

{

.....

schedule（）；
```



```

if (tmp)

tmp->state=0; //将tmp所对应的进程设置为就绪态

}

```

我们来看图7-31。

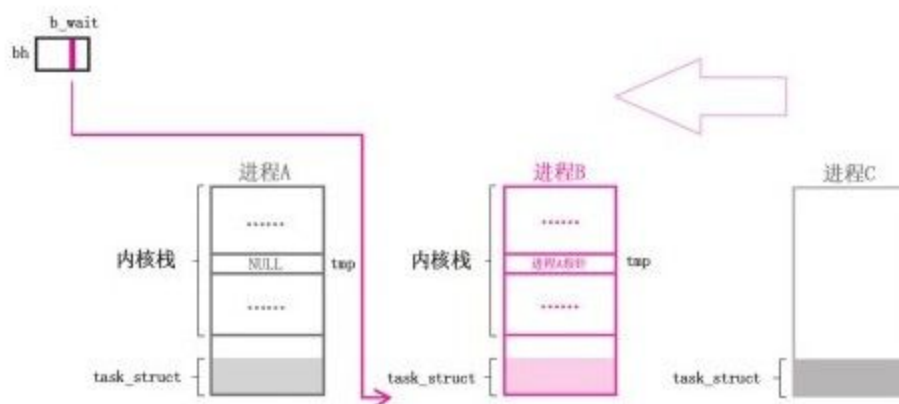


图 7-31 进程B被唤醒，退出进程等待队列

此时内核中程序在执行，所使用的是进程B的内核栈，这样tmp对应的是进程A的task_struct指针，所以此时是将进程A设置为就绪态。

唤醒进程A的情景如图7-32所示。

当前进程为进程**B**，将缓冲块中指定的数据读出，执行代码如下：

//代码路径： fs/file_dev.c:

```
int file_read (struct m_inode * inode,struct file * filp,char * buf,int
count)
```

```
{
```

```
.....
```

```
if (bh) {
```

```
char * p=nr+bh->b_data;
```

```
while (chars-->0)
```

```
put_fs_byte (* (p++) , buf++) ; //将数据读入进程B用户空间
```

```
brelse (bh) ;
```

```
}else{
```

```
while (chars-->0)
```

```
put_fs_byte (0, buf++) ;
```

```
}
```

.....

}

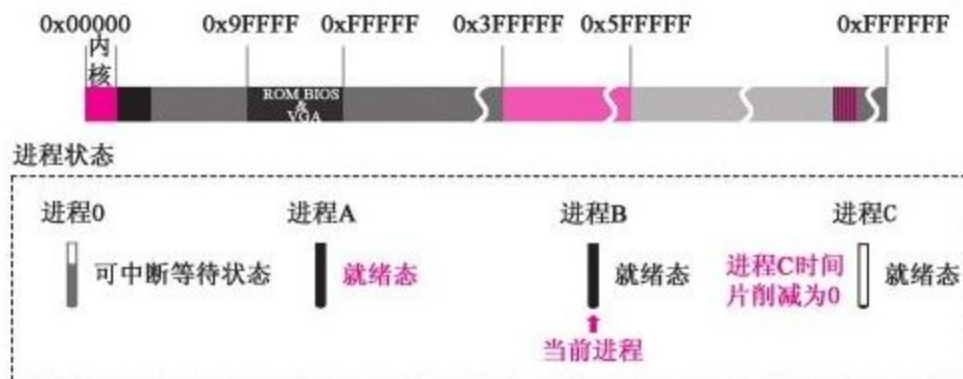


图 7-32 唤醒进程A

之后回到进程B的程序中执行如下代码：

```
for (i=0; i<1000000; i++) //消耗时间片
{
    for (j=0; j<1000000; j++)
    {
        ;
    }
}
```

}

随着时钟中断的不断产生，进程B的时间片削减为0后，由于此时系统中只有进程A处于就绪态，而且它的时间片没有被削减为0，所以就会切换到进程A去执行，如图7-33所示。

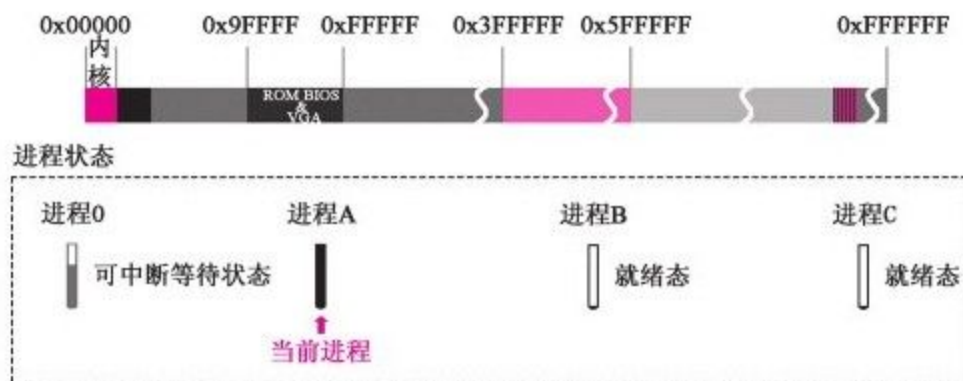


图 7-33 切换到进程A执行

进程A也是在sleep_on () 函数中，调用了schedule () 函数进行的进程切换，因此，进程A

最终回到sleep_on () 函数中，首先要执行的代码如下：

```
//代码路径: kernel/sched.c:
```

```
void sleep_on (struct task_struct ** p)
```

```
{
```

```
.....
```

```
schedule ();
```

```
if (tmp) //此时tmp为NULL
```

```
tmp->state=0; //这里代码不执行，不再唤醒进程了
```

```
}
```

此次执行的情景不太一样，我们来看图7-

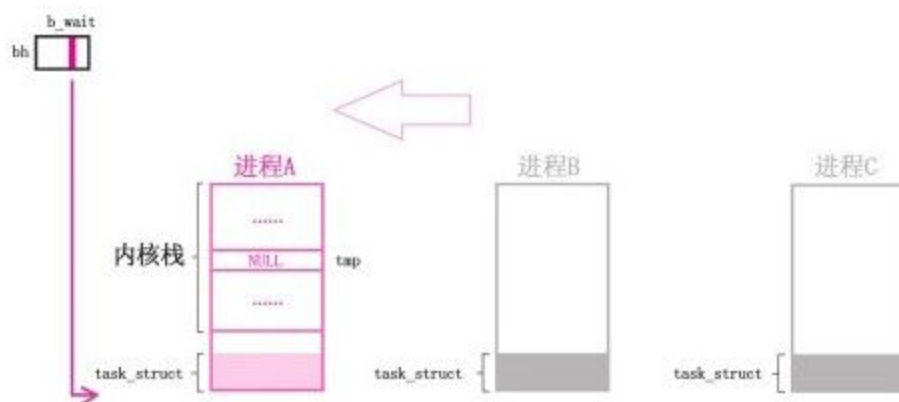


图 7-34 进程A被唤醒，退出进程等待队列

此时内核中程序在执行，所使用的是进程A的内核栈，这样tmp对应的是NULL，不会再唤醒进程了。

以上就是进程等待队列中，进程被唤醒的过程。这三个进程挂起的顺序依次为进程A、进程B、进程C。前面介绍的唤醒顺序为进程C、进程B、进程A，刚好与挂起的顺序相反。

7.7 总体来看缓冲块和请求项

`b_dev`、`b_blocknr`是缓冲区中数据停留的标志。在对缓冲块的实际应用中，内核并没有刻意清除这两个字段。这意味着，如果持续新申请缓冲块，那么很快所有的缓冲块就都会与数据块绑定。这时再申请缓冲块，就只能用新的绑定关系替换旧的绑定关系，这个缓冲块中已有的数据作废。这体现了让缓冲区中的数据在缓冲区中停留的时间尽可能长的策略。

为了能够让数据多停留一段时间，内核要做到尽可能地不申请新的缓冲块，能沿用已经建立绑定关系的缓冲块就沿用，实在不行了，非得申

请不可了，再去申请。这一做法在代码中的体现如下：

```
//代码路径: fs/buffer.c:

struct buffer_head * getblk (int dev,int block) //申请缓冲块

{

repeat:

    if (bh=get_hash_table (dev,block) ) //如果发现缓冲块与指定设备 (dev) 指定数据块 (block) 已经绑定

return bh; //直接用现成的

    tmp=free_list; //如果没找到符合指定标准的绑定缓冲块，再说申请新缓冲块

do{

.....

/*and repeat until we find something good*/

}while ( (tmp=tmp->b_next_free) !=free_list) ;

.....

}
```

从以上代码中不难发现，内核先搜索哈希表，通过比对**b_dev**、**b_blocknr**，来分析是否可以沿用，如果可以，直接返回使用就可以，实在不行，再执行**do.....while**循环，新申请缓冲块。

下面我们来看新申请缓冲块时的情景。

代码如下：

```
//代码路径： fs/buffer.c:

#define BADNESS (bh) ( ( (bh) -> b_dirt << 1) + (bh) ->
b_lock)

struct buffer_head * getblk (int dev,int block) //申请缓冲块
{
.....

tmp=free_list;

do{
```

```

if (tmp->b_count) //如果缓冲块是被占用的，直接跳过本次循环
continue;

if (! bh||BADNESS (tmp) < BADNESS (bh) ) { //权衡
BADNESS，选择缓冲块

bh=tmp;

if (! BADNESS (tmp) )

break;

}

/*and repeat until we find something good*/

}while ( (tmp=tmp->b_next_free) !=free_list) ;

.....

}

```

现在的情况是，哈希表中已经遍历过了，缓冲区中所有缓冲块都不能被进程沿用了，内核必须申请一个新缓冲块。申请的时候，从free_list表头开始搜索，这是为了尽可能不破坏已经绑定数

据块的缓冲块，让它们多停留在缓冲区一会儿，实在不行了（比如缓冲区中所有缓冲块都和数据块绑定了），那就只好用新关系替换老关系了。

循环里面的执行，是在这一前提下开始的。

循环里面并没有分析**b_uptodate**字段。这是因为，既然前面搜索哈希表时已经确认，没有合适的可以沿用的缓冲块了，那么这就意味着，对于当前进程而言，缓冲区中所有缓冲块的内容已经不可用了，它们是不是更新了，**b_uptodate**是1还是0，都无所谓，所以此时也就无须分析**b_uptodate**的数值了。

循环中首先判断**b_count**是否为0。如果不为0，说明缓冲块正在被其他进程共享，当前进程不

能废除正在被其他进程共享的缓冲块，这个缓冲块不能用，内核得为它再新建缓冲块。如果找遍缓冲区也找不到b_count为0的缓冲块，那么当前进程只能挂起了。

代码如下：

```
//代码路径： fs/buffer.c:

#define BADNESS (bh) ( ( (bh) ->b_dirt<<1) + (bh) ->
b_lock)

struct buffer_head * getblk (int dev,int block) //申请缓冲块
{
.....

tmp=free_list;

do{

if (tmp->b_count) //如果缓冲块是被占用的，直接跳过本次循环

continue;
```

```

if ( ! bh || BADNESS (tmp) < BADNESS (bh) ) {

bh=tmp;

if ( ! BADNESS (tmp) )

break;

}

/*and repeat until we find something good*/

}while ( (tmp=tmp->b_next_free) != free_list) ;

if ( ! bh) { //最终也没找到b_count为0的缓冲块

sleep _on (&buffer_wait) ; //当前进程只好挂起

goto repeat;

}

.....

}

```

如果找到了b_count为0的缓冲块，还有两个字段会左右进一步的选择。一个是b_dirt，另一个

是**b_lock**。如果这两个字段都为0，选择这样的缓冲块再合适不过了，可以直接使用。如果**b_lock**和**b_dirt**中有一个是1，那么选择哪个合适呢？相比来讲，选择**b_lock**为1的更为有利。理由是，这两个字段有一个为1，当前进程肯定都不能使用了，肯定要等，相比之下等的时间当然越少越好。**b_lock**为1，说明该缓冲块正在跟硬盘交互数据，交互完了，最终轮到当前进程使用。而**b_dirt**为1，说明在建立新的绑定关系之前，肯定需要把数据同步到硬盘，同步的时候肯定要加锁——**b_lock**置1。所以，选择**b_lock**为1的比选择**b_dirt**为1的，少等待由**b_dirt**为1到**b_lock**为1的时间。这一点从如下代码中也可以看出来。代码如下：

//代码路径：fs/buffer.c:

```

#define BADNESS (bh) ( ( (bh) -> b_dirt < 1) + (bh) ->
b_lock)

struct buffer_head * getblk (int dev,int block) //申请缓冲块

{

.....

tmp=free_list;

do{

if (tmp->b_count) //如果缓冲块是被占用的，直接跳过本次循环

continue;

if ( ! bh||BADNESS (tmp) < BADNESS (bh) ) {

bh=tmp;

if ( ! BADNESS (tmp) )

break;

}

/*and repeat until we find something good*/

}while ( (tmp=tmp->b_next_free) !=free_list) ;

if ( ! bh) { //最终也没找到b_count为0的缓冲块

```

```

sleep_on (&buffer_wait) ; //当前进程只好挂起

goto repeat;

}.....

while (bh->b_dirt) { //如果申请到b_dirt为1的缓冲块，直接写
盘。写完了，当前进程再使用

sync_dev (bh->b_dev) ;

wait_on_buffer (bh) ;

if (bh->b_count)

goto repeat;

}.....

}

```

可见，先不去申请b_dirt为1的缓冲块，更能让进程尽快执行，对它更有利，所以，#define BADNESS (bh) (((bh) ->b_dirt << 1) + (bh) ->b_lock) 中，将b_dirt左移一位，使之权重更高。BADNESS (tmp) < BADNESS (bh)

这行逻辑，将在b_dev、b_blocknr、b_count同等条件下，使b_dirt尽可能靠后被申请到。

7.8 实例2：多进程操作文件的综合实例

下面我们通过一套多进程操作文件的案例，对缓冲块的选择以及请求项的使用等进行介绍。

进程A是一个写盘进程，目的是往hello1.txt文件中写入str1[]中的字符“ABCDE”，代码如下：

```
void FunA () ;

void main ()

{

.....

FunA () ;

.....

}
```

```
void FunA ()  
  
{  
  
char str1[]="ABCDE";  
  
int i;  
  
//打开文件  
  
int fd=open ("/mnt/user/user1/user2/hello1.txt", O_RDWR,  
0644) ) ;  
  
for (i=0; i<1000000; i++)  
  
{  
  
//写文件  
  
write (fd,str1, strlen (str1) ) ;  
  
}  
  
//关闭文件  
  
close (fd) ;  
  
return;  
  
}
```

进程B是一个写盘进程，目的是往hello2.txt文件中写入str1[]中的字符“ABCDE”，代码如下：

```
void FunB () ;

void main ()

{

.....

FunB () ;

.....

}

void FunB ()

{

char str1[]="ABCDE";

int i;

//打开文件

int fd=open ("/mnt/user/user1/user2/hello2.txt", O_RDWR,
0644) ) ;
```

```
for (i=0; i<1000000; i++)  
  
{  
  
//写文件  
  
write (fd,str1, strlen (str1) ) ;  
  
}  
  
//关闭文件  
  
close (fd) ;  
  
return;  
  
}
```

进程C是一个读盘进程，目的是从hello3.txt文件中读20 000字节到buffer中，代码如下：

```
void FunC () ;  
  
void main ()  
  
{  
  
.....
```

```
FunC ( ) ;
```

```
.....
```

```
}
```

```
void FunC ( )
```

```
{
```

```
char buffer[20000];
```

```
int i,j;
```

```
//打开文件
```

```
int fd=open ( "/mnt/user/user1/user2/hello3.txt", O_RDWR,  
0644) ) ;
```

```
//读文件
```

```
read (fd,buffer,sizeof (buffer) ) ;
```

```
//关闭文件
```

```
close (fd) ;
```

```
return;
```

```
}
```

这三个进程的执行顺序为：进程A先执行，之后进程B执行，最后进程C执行。这三个进程没有父子关系。

1.系统不断为进程A向缓冲区写入数据

进程A开始执行后，执行write函数。假设hello1.txt文件没有任何内容，所以进程A只需在缓冲区中不断申请缓冲块并将指定的数据写入缓冲块就可以了。新申请缓冲块的前提是，该缓冲块空闲且不脏。我们假设现在系统已经在缓冲区中的所有空闲且不脏的缓冲块内写满了数据。图7-35显示了系统已经将所有空闲且不脏的缓冲块写满的状态。接下来，我们就要看一下，在缓冲区处于图7-35所示的状态时，再新申请缓冲块并进行写入操作，会导致什么情况。

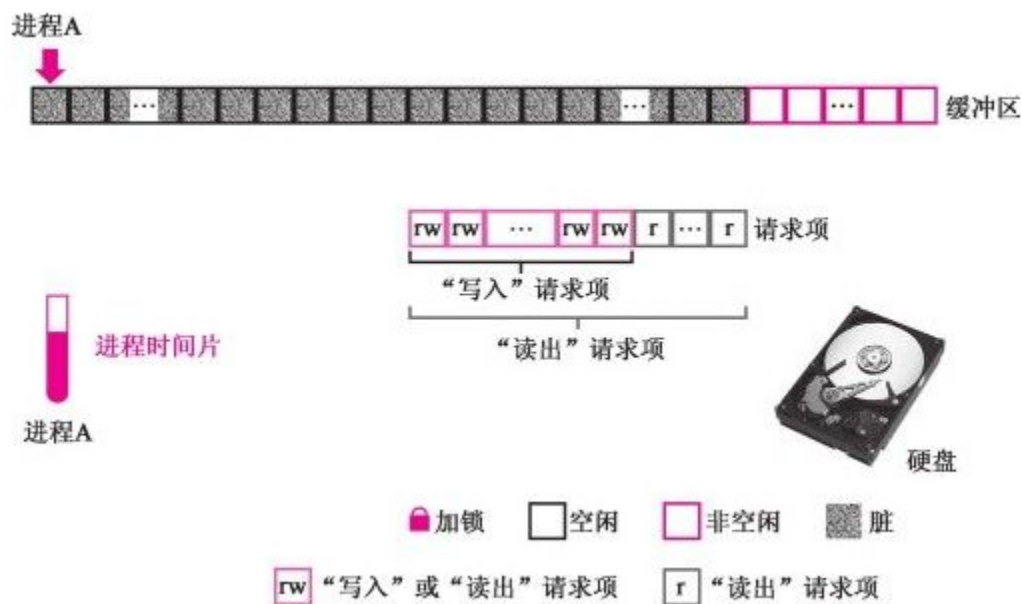


图 7-35 系统不断为进程A写入数据

2.继续执行引发缓冲块数据需要同步

当前进程仍然是进程A。它提出的请求，系统还远没有完成，还要继续向缓冲区写数据。这就需要通过`getblk`函数在缓冲区中找到一个空闲的缓冲块，即`b_count`为0的缓冲块。

执行代码如下：

//代码路径: fs/buffer.c:

```
struct buffer_head * getblk (int dev,int block)
```

```
{
```

```
.....
```

```
tmp=free_list;
```

```
do{
```

```
if (tmp->b_count) //找空闲的缓冲块
```

```
continue;
```

```
if (! bh||BADNESS (tmp) < BADNESS (bh) ) { //在空闲的基础上, 权衡BADNESS
```

```
bh=tmp;
```

```
if (! BADNESS (tmp) )
```

```
break;
```

```
}
```

```
/*and repeat until we find something good*/
```

```
}while ( (tmp=tmp->b_next_free) !=free_list) ;
```

```
.....
```

```
}
```

但现在的情况是，缓冲区中已经没有空闲且不脏的缓冲块，只有空闲且脏的缓冲块。这意味着，接下来要强行将缓冲区中的数据同步到硬盘，以便在缓冲区中空出更多的空间，为后续的写盘工作提供支持，执行代码如下：

```
//代码路径： fs/buffer.c:
```

```
struct buffer_head * getblk (int dev,int block)
```

```
{
```

```
.....
```

```
if (bh->b_count)
```

```
goto repeat;
```

```
while (bh->b_dirt) { //如果申请到的空闲缓冲块都是脏的，说明  
缓冲区里面有太多的数据需要往硬盘同步了
```

```
sync_dev (bh->b_dev) ; //立即同步
```

```
wait_on_buffer (bh) ;  
  
if (bh->b_count)  
  
goto repeat;  
  
}  
  
.....  
  
}
```

3.将缓冲区中的数据同步到硬盘上

此时，`sync_dev ()` 函数用来将缓冲区中的数据同步到硬盘上。进入`sync_dev ()` 函数后，代码如下：

```
//代码路径： fs/buffer.c:  
  
int sync_dev (int dev)  
  
{  
  
.....
```

```

bh=start_buffer;

for (i=0; i<NR_BUFFERS; i++, bh++) { //全部都要遍历

if (bh->b_dev !=dev)

continue;

wait_on_buffer (bh) ;

if (bh->b_dev==dev&&bh->b_dirt) //只要设备号符合是脏的,
就同步

ll_rw_block (WRITE,bh) ;

}

.....

}

```

sync_dev函数将会遍历整个缓冲区，将所有“脏”的缓冲块中的内容全部同步到硬盘上。每个“脏”缓冲块的同步步骤是这样的：

第一，先将缓冲块与申请到的空闲请求项进行绑定，请求项中记录的内容将作为数据同步的唯一依据。

第二，如果硬盘此时没有工作，则下达写盘命令，将数据进行同步；如果硬盘正在工作，则将该请求项挂接在请求项队列中，硬盘同步完数据并产生中断后，中断服务程序会不断地给硬盘发送指令，以使请求项队列中各个请求项对应的数据陆续同步到硬盘中。

`sync_dev`函数将会不停地执行上述工作，直到无法再申请到空闲请求项为止。

每个缓冲块同步的过程都是在`ll_rw_block`函数中完成的。在此过程中，缓冲块会被加锁。加

锁只是阻止进程与缓冲块的数据交互，阻止系统自身与缓冲块的数据交互，但并不阻止缓冲块与硬盘之间的数据交互。在发送同步指令之前，需要同步的缓冲块的脏标志**b_dirt**将会被设置为0，以表示它不再是“脏”的缓冲块了。

具体的执行路线是，进入**ll_rw_block**函数后会调用**make_request**函数将缓冲块与请求项挂接；在**make_request**函数中会先将缓冲块加锁，并通过**add_request**函数加载请求项；在请求项加载完毕后，系统将通过调用**do_hd_request**函数向硬盘发送写盘命令。**do_hd_request**函数是系统与硬盘交互的底层函数，它将根据请求项的数据，最终实现将指定的缓冲块的数据写到指定的硬盘块上。执行代码如下：

//代码路径: kernel/blk_drv/ll_rw_blk.c:

```
static void make_request (int major,int rw,struct buffer_head * bh)

{

.....

if (rw!=READ&&rw!=WRITE)

panic ("Bad block dev command,must be R/W/RA/WA") ;

lock_buffer (bh) ; //给缓冲块加锁

if ( (rw==WRITE&&! bh->b_dirt) || (rw==READ&&bh->
b_uptodate) ) {

unlock_buffer (bh) ;

return;

}

.....

req->buffer=bh->b_data;

req->waiting=NULL;

req->bh=bh;

req->next=NULL;
```

```

    add_request (major+blk_dev,req) ; //加载请求项

}

static void add_request (struct blk_dev_struct * dev,struct request *
req)

{

.....

if (req->bh)

req->bh->b_dirt=0; //缓冲块同步了，就不脏了

.....

(dev->request_fn) () ; //这行代码对应的就是do_hd_request函
数

.....

}

```

同步一个缓冲块的情况如图7-36所示。在make_request函数中把这个缓冲块加锁了，而在add_request函数中已经将该缓冲块的脏标志置0，

此时这个缓冲块已经成为了空闲且不脏的缓冲块。请读者将图7-36与图7-35对比，注意该缓冲块的状态变化。

<p>

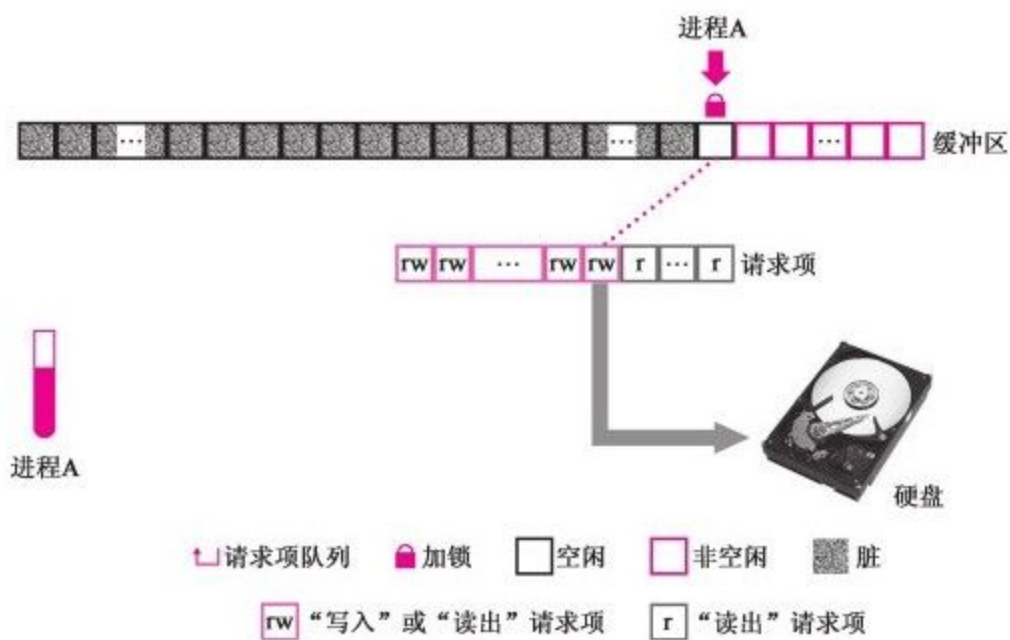


图 7-36 将写请求插入请求项队列

`sync_dev`函数不断同步缓冲块，最终结果如图7-37所示。注意所有留给写入操作的请求项均

已被占用，同时与写入请求项对应的缓冲块的状态也已被置为空闲且不脏的状态，而硬盘正在不断对请求项进行处理。

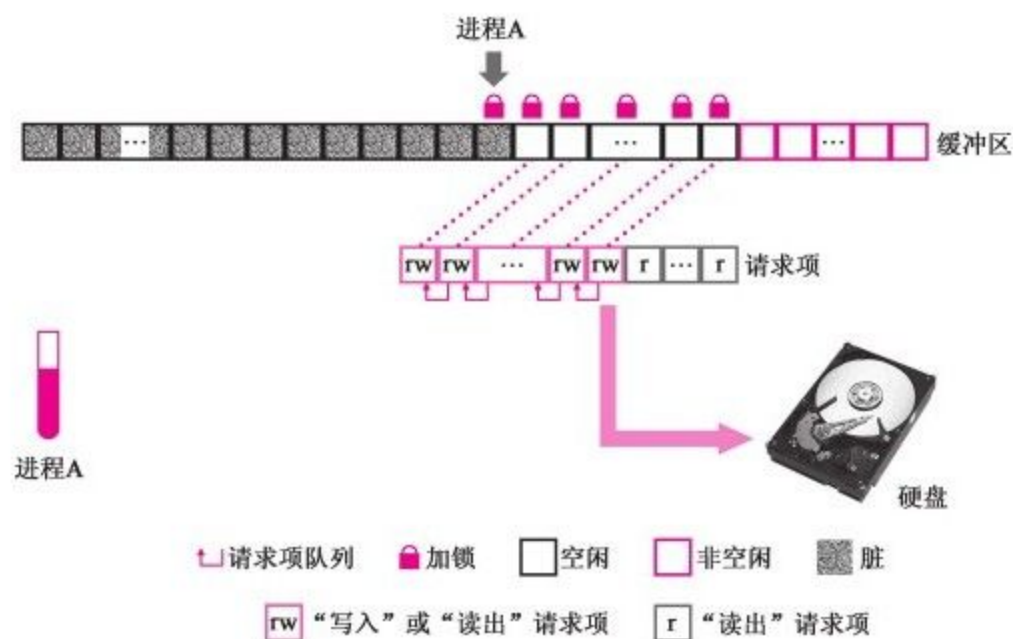


图 7-37 请求项结构中供写请求使用的空间已用完

实现此过程的代码如下：

```
//代码路径： fs/buffer.c:
```

```

int sync_dev (int dev)

{

int i;

struct buffer_head * bh;

bh=start_buffer;

for (i=0; i<NR_BUFFERS; i++, bh++) { //遍历所有缓冲块

if (bh->b_dev != dev)

continue;

wait_on_buffer (bh) ;

if (bh->b_dev==dev && bh->b_dirt)

ll_rw_block (WRITE,bh) ;

}

.....

}

```

//代码路径: kernel/blk_drv/ll_rw_blk.c:

```

void ll_rw_block (int rw,struct buffer_head * bh)

{

```

```

unsigned int major;

if ( (major=MAJOR (bh->b_dev) ) >=NR_BLK_DEV||

! (blk_dev[major].request_fn) ) {

printk ("Trying to read nonexistent block-device\n\r") ;

return;

}

make_request (major,rw,bh) ;

}

static void make_request (int major,int rw,struct buffer_head * bh)

{

.....

if (rw!=READ&&rw!=WRITE)

panic ("Bad block dev command,must be R/W/RA/WA") ;

lock_buffer (bh) ; //这里加锁了

.....

add_request (major+blk_dev,req) ;

}

```

```

static void add_request (struct blk_dev_struct * dev,struct request *
req)

{

.....

req->next=NULL;

cli () ;

if (req->bh)

req->bh->b_dirt=0; //这里把脏标志置为0

.....

}

```

请求项结构中虽然还有空闲的请求项，但留给“写入”操作的请求项只占请求项总数的2/3，对应的代码如下：

//代码路径： kernel/blk_drv/ll_rw_blk.c:

```

static void make_request (int major,int rw,struct buffer_head * bh)

```

```

{

.....

if (rw==READ)

req=request+NR_REQUEST; //所有请求项都可以用来读操作

else

    req=request+ ((NR_REQUEST*2)/3); //只有2/3的请求项可以用来写操作

.....

}

```

由于现在这2/3的请求项已经全部被占用了，所以现在已经没有空闲的请求项为“同步”服务了，如图7-35中的写入请求项所示。

4.进程A由于等待空闲请求项而被系统挂起

空闲的“写盘”请求项没有了，但sync_dev

() 函数仍然会被继续调用，再次进入

make_request () 函数后，将会执行如下代码：

```
//代码路径: kernel/blk_drv/ll_rw_blk.c:
```

```
static void make_request (int major,int rw,struct buffer_head * bh)
```

```
{
```

```
.....
```

```
while (--req >= request)
```

```
.....
```

```
if (req < request) { //执行到这里，说明没有空闲的请求项了
```

```
.....
```

```
sleep_on (&wait_for_request) ;
```

```
}
```

```
.....
```

```
}
```

这些代码的作用是，如果最后也没有找到合适的空闲请求项，就将当前进程挂起。调用 `sleep_on()` 函数后，进程A就成为了等待空闲请求项的进程，被挂起了。该过程如图7-38所示，此时硬盘仍然在不断地处理请求项，而进程A已经处于挂起状态，尽管它还有时间片。

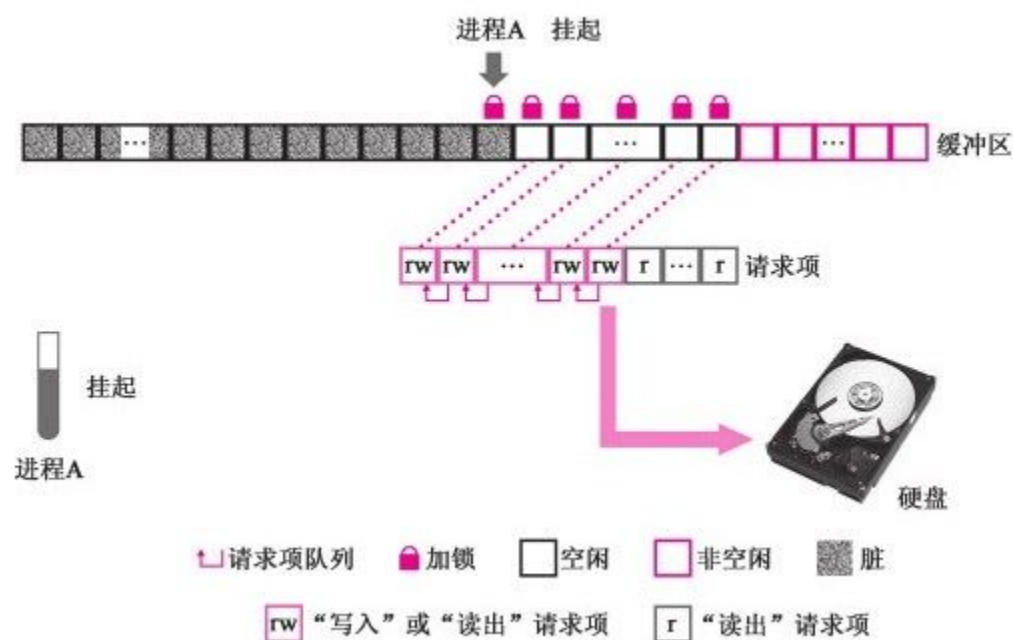


图 7-38 进程A被挂起

5.进程B开始执行

进程B开始执行。它也是一个写盘进程。系统也要给进程B申请缓冲块，以便其写入数据。执行代码如下：

```
//代码路径： fs/buffer.c:
```

```
struct buffer_head * getblk (int dev,int block)
```

```
{
```

```
.....
```

```
tmp=free_list;
```

```
do{
```

```
if (tmp->b_count)
```

```
continue;
```

```
if (! bh||BADNESS (tmp) <BADNESS (bh) ) {
```

```
bh=tmp;
```

```
if (! BADNESS (tmp) )
```

```
break;
```

```

}

/*and repeat until we find something good*/

}while ( (tmp=tmp->b_next_free) !=free_list) ;

.....

}

```

通过图7-38可以看出，现在缓冲区中的各个空闲缓冲块的状态有所不同，系统就是要在当前情况下综合分析所有的空闲缓冲块的状态，从而确定为进程B申请哪个缓冲块。系统是通过BADNESS (tmp) 来进行综合分析的。BADNESS (tmp) 的定义如下：

```

#define BADNESS (bh)  ( ( (bh) -> b_dirt
<< 1) + (bh) -> b_lock) ,

```

通过前面的分析我们得知，它的作用是将缓冲区中所有的缓冲块按照对进程执行更有利的原则，分为4个等级，从有利到不利依次为：

一等：没有“脏”，且没有“加锁”的空闲缓冲块，此等缓冲块的BADNESS值为0；二等：没有“脏”，且有“加锁”的空闲缓冲块，此等缓冲块的BADNESS值为1；

三等：有“脏”，且没有“加锁”的空闲缓冲块，此等缓冲块的BADNESS值为2；

四等：有“脏”，且有“加锁”的空闲缓冲块，此等缓冲块的BADNESS值为3。

BADNESS值越小，则该缓冲块就越便于使用；越大就越不便于使用。

系统已经将一些缓冲块加锁，并将它们的“脏”标志设置为0了，于是就让它们的BADNESS值为1。在目前的情况下，这已经是最便于使用的缓冲块了。因此系统就为进程B申请到了一个BADNESS值为1的缓冲块。这个缓冲块是加锁的。这意味着该缓冲块并不能马上被操作，但总比申请一个脏的缓冲块好得多。图7-39表示了系统给进程B申请到的缓冲块。

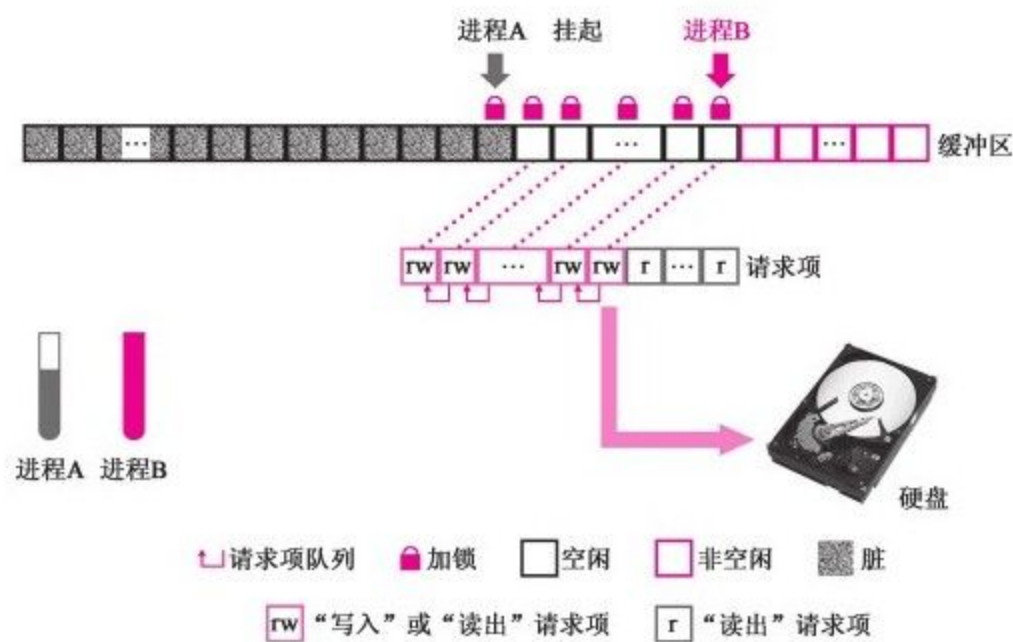


图 7-39 系统为进程B申请到缓冲块

6.进程B也被挂起

系统申请到的缓冲块是“加锁”的缓冲块，这就导致进程或系统不能立即与该缓冲块进行数据交互。于是，系统会直接调用wait_on_buffer（）函数，进程B将会被挂起，如图7-40所示。注意，此时系统和硬盘还在不断处理请求项。

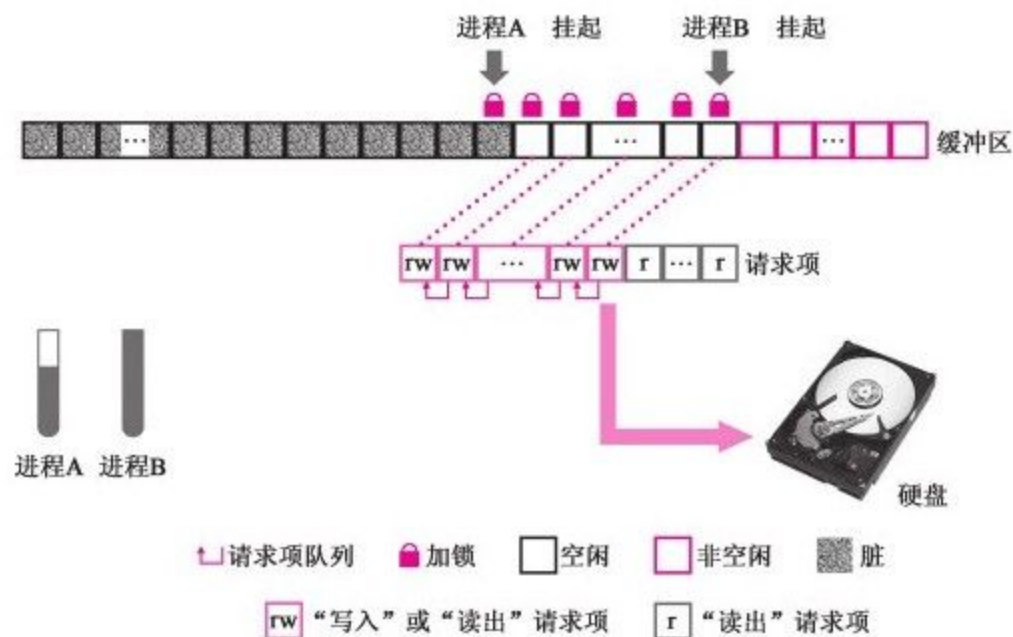


图 7-40 进程B也被挂起

执行代码如下：

```
//代码路径: fs/buffer.c:

struct buffer_head * getblk (int dev,int block)

{

.....

if (! bh) {

sleep_on (&buffer_wait) ;

goto repeat;

}

wait_on_buffer (bh) ; //这里将进程B挂起

if (bh->b_count)

goto repeat;

.....

}
```

7.进程C开始执行并随后被挂起

进程C开始执行。它是一个读盘进程。系统也要为它申请一个缓冲块。从现在缓冲区的情况来看，系统给进程C和进程B申请的应该是同一个缓冲块。前面提到的缓冲块是加锁的，该缓冲块同样不能被操作，但可以被申请到，于是进程C也会被挂起。

进程C和进程B现在都因为在等待同一个缓冲块的解锁而被挂起，所以这两个进程现在就形成了一个进程等待队列。

到现在为止，实例2中的三个用户进程都被挂起了，于是默认切换到进程0去执行。它们被挂起的情况如图7-41所示。其中进程A处在等待空闲请

求项的等待队列，而进程B和进程C处在等待同一个缓冲块解锁的队列。

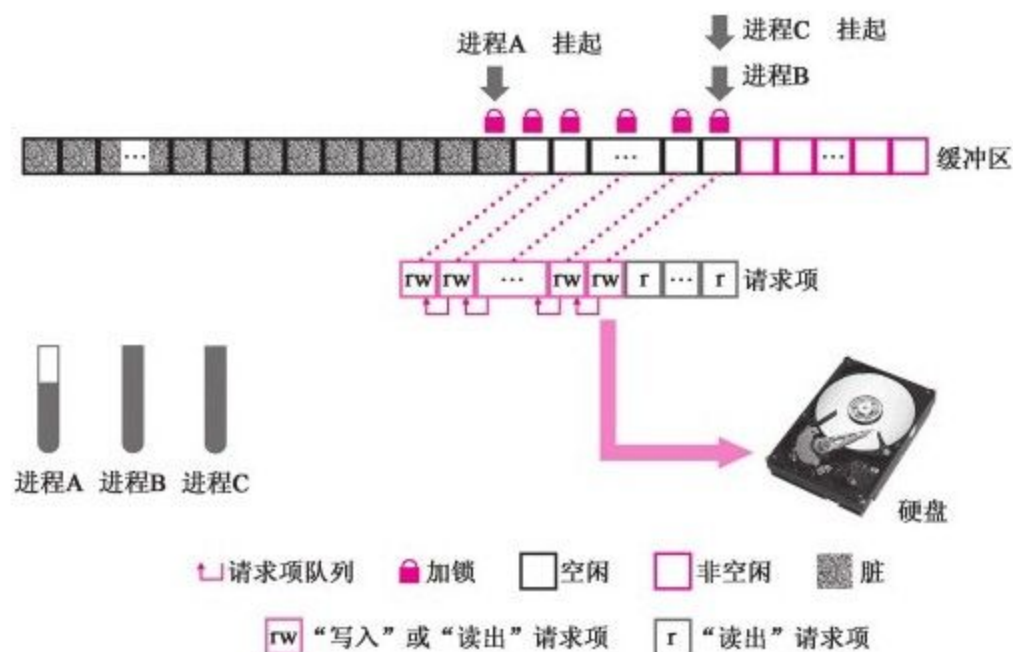


图 7-41 进程A、B和C此时的运行状态和所处等待队列

8.进程A和进程C均被唤醒

下面我们将介绍这三个用户进程被唤醒的过程。它们被唤醒后，系统将继续根据缓冲块和请

求项的各方面状况来决定用户进程将如何执行。

一段时间之后，硬盘完成了请求项交付的同步工作，产生硬盘中断，中断服务程序开始执行，执行代码如下：

```
//代码路径: kernel/blk_dev/blk.h:
```

```
extern inline void end_request (int uptodate)
```

```
{
```

```
.....
```

```
unlock_buffer (CURRENT->bh) ;
```

```
.....
```

```
wake_up (&wait_for_request) ;
```

```
.....
```

```
}
```

```
extern inline void unlock_buffer (struct buffer_head * bh)
```

```
{
```

```
.....
```

```
bh->b_lock=0;
```

```
wake_up (&bh->b_wait) ;
```

```
}
```

进程A是由于等待空闲请求项才被挂起的，于是“wake_up (&wait_for_request) ; ”这行代码将唤醒进程A，如图7-42所示。

进程B和进程C构成了一个进程等待队列。进程C后挂起，要先唤醒，如图7-42所示。

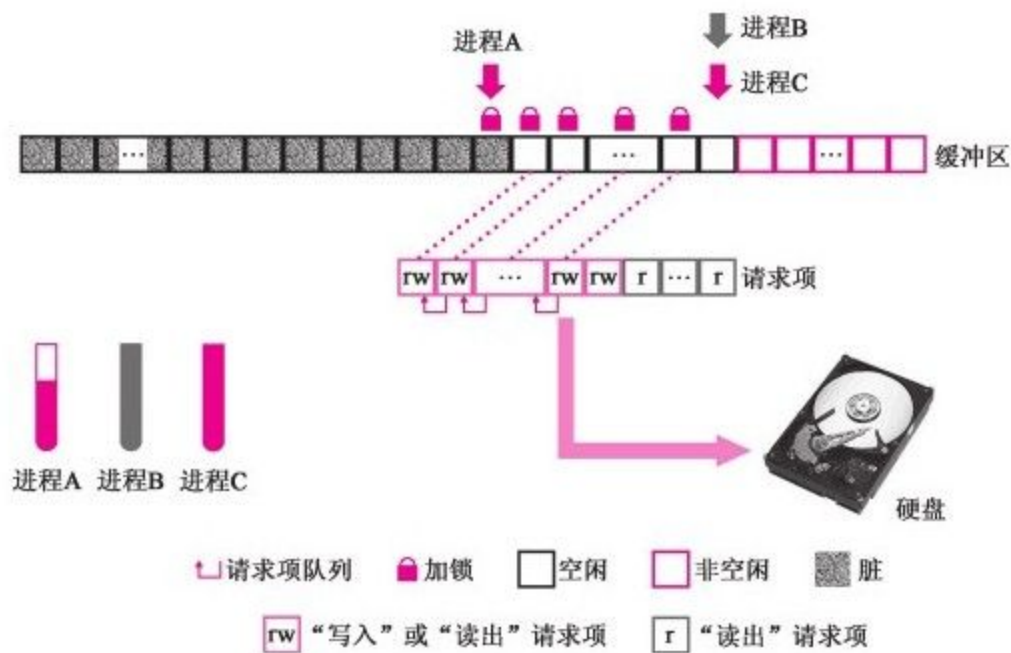


图 7-42 一个缓冲块数据同步完成后各进程的状态

另外，由于指定的缓冲块中的数据已经操作完毕了，所以中断服务程序还会将该缓冲块解锁。中断服务程序会在以上工作做完后继续调用 `do_hd_request` 函数，如果还有请求项需要处理，再次给硬盘发送写盘指令。很显然，硬盘又要开始进行后续的同步工作了。从图7-42中可以看

出，硬盘已经在处理下一个请求项，而此刻已经有一个可用的“写”请求项空闲。

此时进程C的时间片多于进程A的时间片，所以系统将当前进程由进程0切换到进程C，开始执行后的第一件事就是要唤醒进程B，如图7-42所示。执行代码如下：

```
//代码路径: fs/buffer.c:

struct buffer_head * getblk (int dev,int block)

{

.....

if (! bh) {

sleep_on (&buffer_wait) ;

goto repeat;

}
```

`wait_on_buffer (bh) ;` //进程C唤醒后从这里开始继续执行，先唤醒进程B

`if (bh->b_count)`

`goto repeat;`

`.....`

`}`

`static inline void wait_on_buffer (struct buffer_head * bh)`

`{`

`cli () ;`

`while (bh->b_lock)`

`sleep_on (&bh->b_wait) ;`

`sti () ;`

`}`

系统为进程C申请到的缓冲块现在已经被解锁了，可以使用了。系统将先对这个缓冲块进行设置，包括将它的引用计数设置为1，代码如下：

//代码路径: fs/buffer.c:

```
struct buffer_head * getblk (int dev,int block)
```

```
{
```

```
.....
```

```
if (! bh) {
```

```
sleep_on (&buffer_wait) ;
```

```
goto repeat;
```

```
}
```

```
wait_on_buffer (bh) ; //唤醒进程B后, 进程C继续从这里执行
```

```
if (bh->b_count)
```

```
goto repeat;
```

```
.....
```

```
bh->b_count=1; //引用计数设置为1
```

```
bh->b_dirt=0;
```

```
bh->b_uptodate=0;
```

```
.....
```

```
}
```

于是该缓冲块就不再是空闲缓冲块了。然后，在请求项结构中申请一个空闲请求项与此缓冲块绑定，并将该缓冲块再次加锁。但是，请求项设置完毕，并不代表马上就能处理这个请求项。此时硬盘正忙着同步处理其他请求项，现在只能先将“读盘”请求项插入请求项队列中，代码如下：

```
//代码路径: kernel/blk_dev/ll_rw_blk.c:

static void add_request (struct blk_dev_struct * dev,struct request *
req)
{
    .....

    for (; tmp->next; tmp=tmp->next) //设备忙，将请求项插入队
列

    if ( (IN_ORDER (tmp,req) ||
```

```
! IN_ORDER (tmp,tmp->next) ) &&
IN_ORDER (req,tmp->next) )

break;

req->next=tmp->next; //next用来组建请求项队列

tmp->next=req;

sti ( ) ;

}
```

然后系统会将进程C挂起。

此时系统中的进程状态如图7-43所示，其中读请求项占据的是请求项数组的最后一项，并且由请求项数组中的第1项的指针指向它。

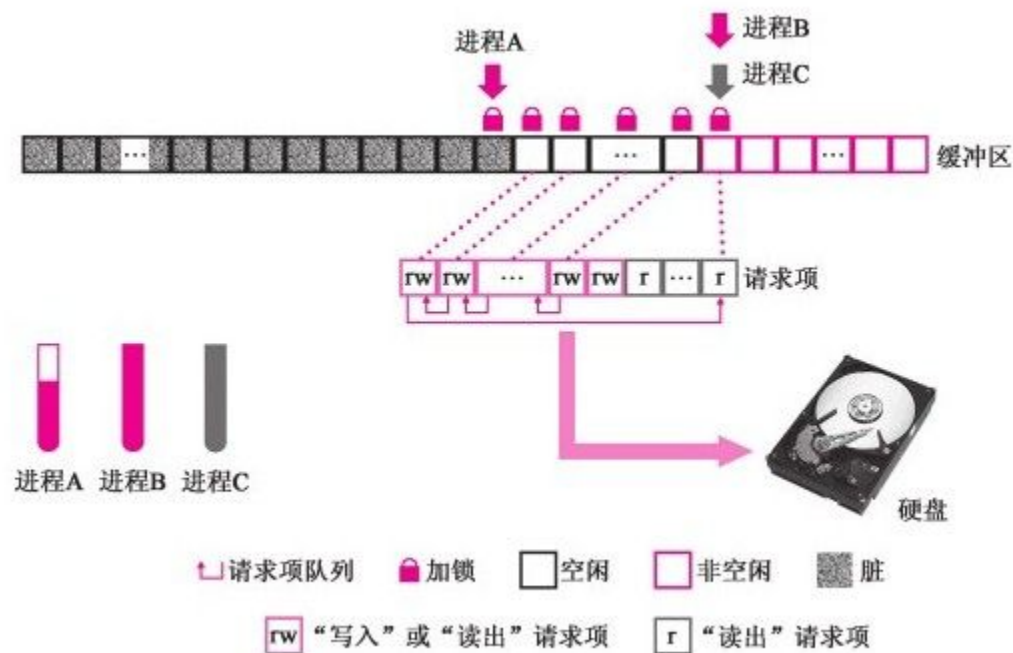


图 7-43 进程C将读请求插入请求项队列后各进程的状态

9.进程B切换到进程A执行

进程C挂起后，进程B的时间片显然比进程A要多，所以切换到进程B。系统早已经为进程B申请了缓冲块，当时由于这个缓冲块是加锁的，所以进程B要挂起。现在，这个缓冲块仍然是加锁

的，所以进程B将再次被挂起，并切换到进程A，如图7-44所示。

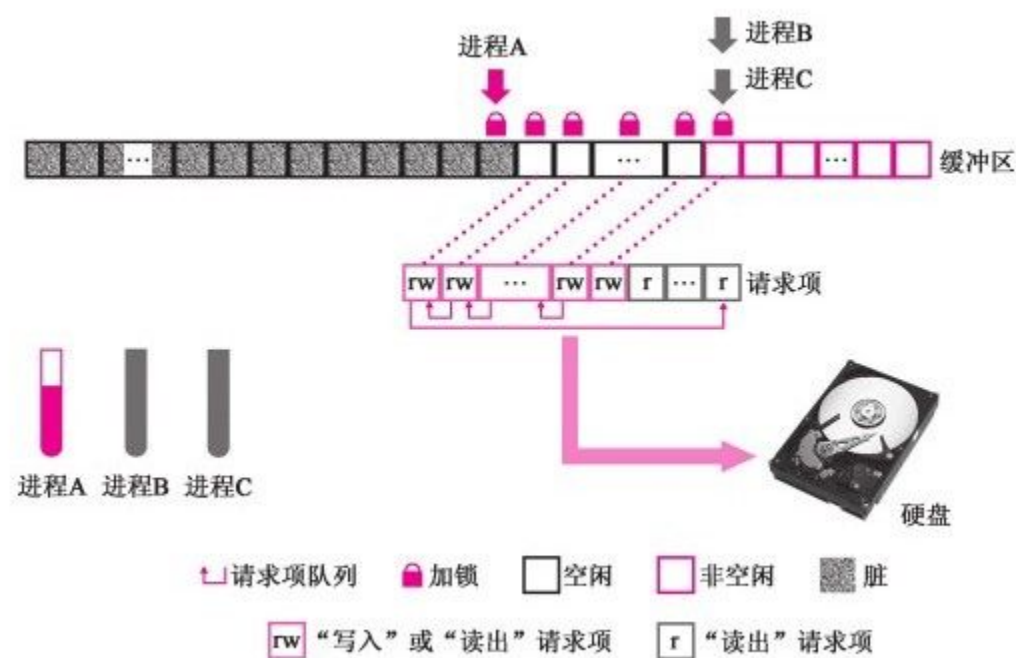


图 7-44 进程B挂起，切换到进程A执行

当前进程是进程A。进程A由于同步缓冲块时缺少空闲请求项才被挂起，所以它被唤醒后，系统将继续同步缓冲块。现在系统已经有空闲的请求项用于写盘了，所以系统将此请求项与即将要

同步的缓冲块绑定，并插入请求项队列中，之后又没有空闲的请求项了，进程A将再次被挂起，如图7-45所示。

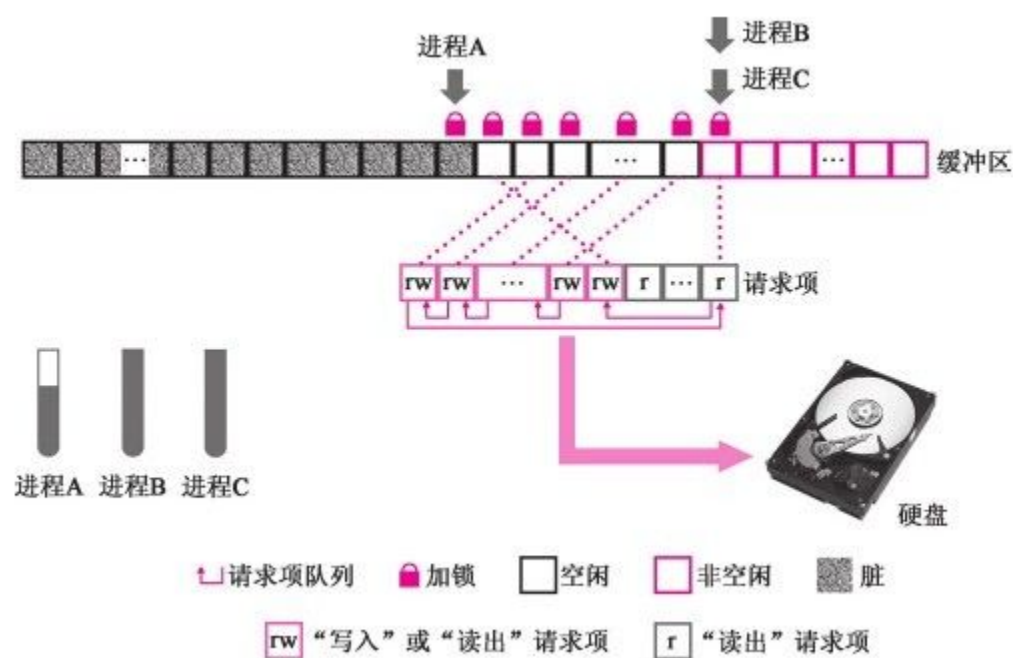


图 7-45 进程A再次挂起

接下来，以上步骤将会重复执行。一方面，只要硬盘执行完一次同步操作，就会释放一个请求项，并将其对应的缓冲块解锁，这些都将导致等待空闲请求项或等待缓冲块解锁的进程被唤

醒；另一方面，被唤醒的进程又不断地引发缓冲区与硬盘之间进行数据交互，从而使这些进程不断地被挂起，如图7-46所示。

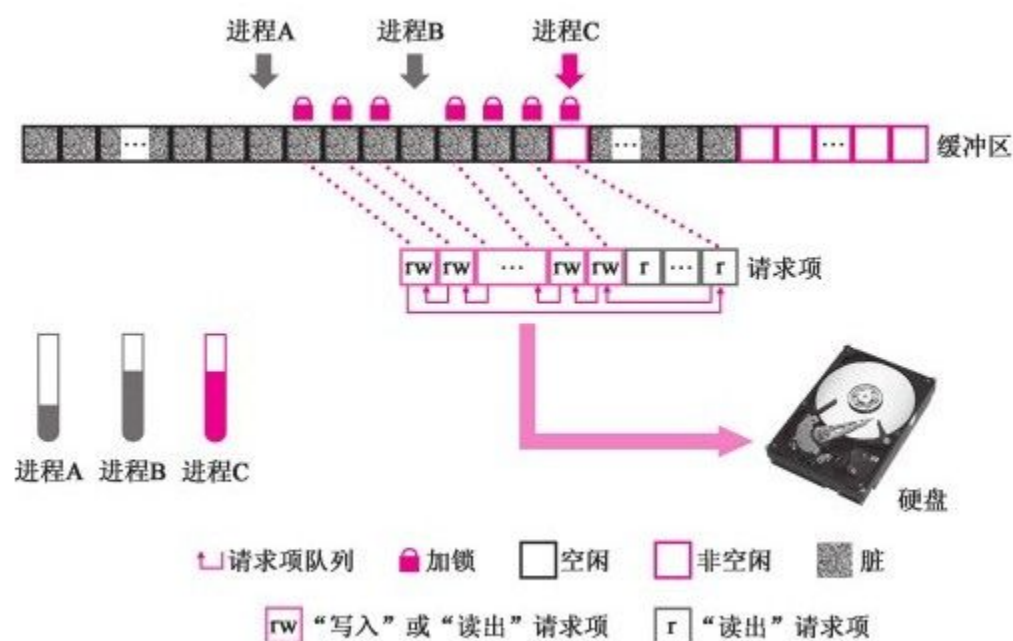


图 7-46 缓冲块再次同步完成时唤醒进程B

直到最后三个进程全部完成各自的写盘任务和读盘任务，如图7-47所示。

理解多进程操作文件的关键是深入理解缓冲区。现在可以看得更清楚，缓冲区的设计指导思想就是让缓冲区中的数据在缓冲区中停留的时间尽可能长，进程与硬盘的交互尽可能在缓冲区中实现，尽可能少地读写硬盘数据。

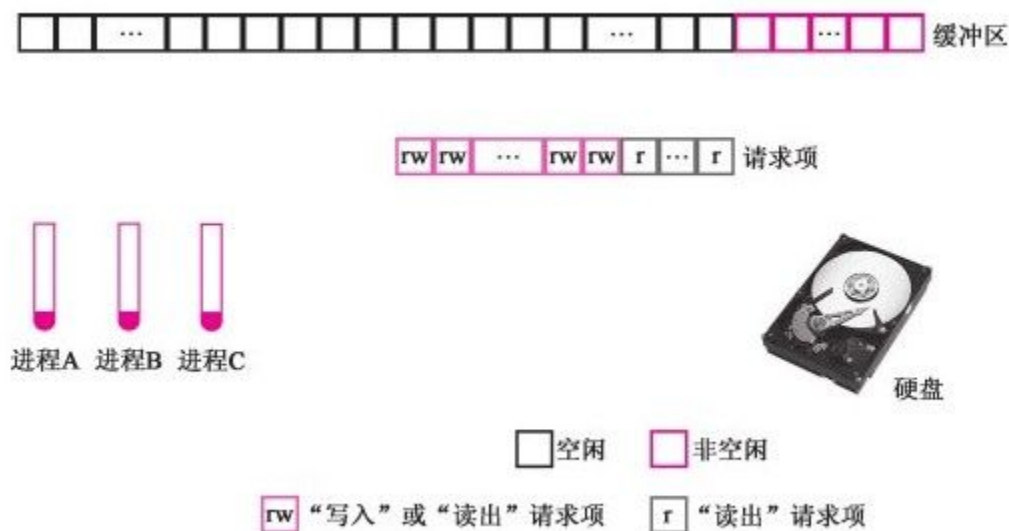


图 7-47 进程所有请求均已处理完毕

从这个设计指导思想出发，仔细审核Linux的源代码，可以看出，同步代码的设计多少与这个设计指导思想有些偏离。

//代码路径: fs/buffer.c:

```
int sys_sync (void)
```

```
{
```

```
int i;
```

```
struct buffer_head * bh;
```

```
sync_inodes () ; /*write out inodes into buffers*/
```

```
bh=start_buffer;
```

```
for (i=0; i<NR_BUFFERS; i++, bh++) { //遍历整个缓冲区,  
不放过一个缓冲块
```

```
wait_on_buffer (bh) ;
```

```
if (bh->b_dirt)
```

```
ll_rw_block (WRITE,bh) ;
```

```
}
```

```
return 0;
```

```
}
```

```
int sync_dev (int dev)
```

```
{
```

```

int i;

struct buffer_head * bh;

bh=start_buffer;

    for (i=0; i<NR_BUFFERS; i++, bh++) { //遍历整个缓冲区,
不放过一个缓冲块

        if (bh->b_dev != dev)

            continue;

        wait_on_buffer (bh) ;

        if (bh->b_dev==dev && bh->b_dirt)

            ll_rw_block (WRITE,bh) ;

    }

    sync_inodes ( ) ;

    bh=start_buffer;

    for (i=0; i<NR_BUFFERS; i++, bh++) { //遍历整个缓冲区,
不放过一个缓冲块

        if (bh->b_dev != dev)

            continue;

        wait_on_buffer (bh) ;

```

```
if (bh->b_dev==dev&&bh->b_dirt)

ll_rw_block (WRITE,bh) ;

}

return 0;

}
```

以上代码中没有考虑**b_count**，不管其是否为0，只要是**b_dirt**为1的缓冲块，都要同步。与尽可能多地共享缓冲区、尽可能少地读写硬盘的设计宗旨不十分相符。

7.9 本章小结

多进程操作文件的核心是多进程对缓冲区的操作。缓冲区关联着用户进程、文件系统和内存，所以本章是比较难的一章。

本章详细讲解了缓冲区的作用及整体架构，全面分析了**b_dev**、**b_blocknr**、**uptodate**、**dirt**.....的作用，并以两个实例详细讲解了进程等待队列及多进程操作文件。

深入理解这一章的内容，对深入理解文件系统、缓冲区、多进程的运行大有益处。

第8章 进程间通信

前面几章讲解了在Linux 0.11中不允许进程跨越边界去访问其他进程的代码、数据，这是操作系统保护模式的核心内容。

从实际应用角度看，进程间往往需要协同工作、交互信息，这似乎与进程保护相违背。如何才能既不破坏进程保护，又能实现进程间通信的合理要求？Linux 0.11设计了两套机制来为此需求提供服务：一套是“管道机制”，另一套是“信号机制”。本章将通过两个实际的应用案例来对这两套机制进行详细的介绍。

8.1 管道机制

为了体现对进程的保护，在不跨越进程边界的前提下实现进程间通信，Linux 0.11绕过对进程边界的保护，设计了管道机制。每个管道允许两个进程交互数据，一个进程向管道中输入数据，一个管道从管道中输出数据，如图8-1所示。该机制实现了进程间通信，同时又不需要非法跨越进程间边界。



图 8-1 管道机制

操作系统在内存中为每个管道开辟一页内存，给这一页内存赋予了文件的属性（赋予文件属性的理由，将在第9章讲解）。这一页内存由两

个进程共享，但不会分配给任何进程，只由内核掌控。

在Linux 0.11中，管道操作分为两部分，一部分是创建管道，另一部分是管道的读写操作。下面我们通过实例1对这两部分内容进行介绍。实例1代码如下：

```
#include <stdio.h>

#include <unistd.h>

int main ()

{

    int n,fd[2];

    pid_t pid;

    int i,j;

    char
str1[]="ABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDE
```

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCDEABCDEABCDEAB
CDEABCDE

ABCDEABCDEABCDEABCDEABCD";

char str2[512];

if (pipe (fd) < 0) { //创建管道

printf ("pipe error\n") ;

return-1;

}

if ((pid=fork ()) < 0) {

```
printf ("fork error\n") ;

return-1;

}

else if (pid > 0) //父进程向管道中写入数据

{

close (fd[0]) ;

for (i=0; i<10000; i++)

write (fd[1], str1, strlen (str1) ) ;

}

else{//子进程从管道中读取数据

close (fd[1]) ;

for (j=0; j<20000; j++)

read (fd[0], str2, strlen (str2) ) ;

}

return 0;

}
```

实例1表现了进程间共享数据的情景：父进程把str1中的数据写入管道，子进程从管道中读出数据，其中str1中字符长度为1024字节，即1 kB。

8.1.1 管道的创建过程

从技术上看，管道就是一页内存，但进程要以操作文件的方式对其进行操作，这就要求这页内存具备一些文件属性并减少页属性。

具备一些文件属性表现为，创建管道相当于创建一个文件，如进程task_struct中*filp[20]和file_table[64]挂接、新建i节点、file_table[64]和文件i节点挂接等工作要在创建管道过程中完成，最终使进程只要知道自己在操作管道类型的文件就可以了，其他的都不用关心。

减少页属性表现为，这页内存毕竟要当做一个文件使用，比如进程不能像访问自己用户空间的数据一样访问它，不能映射到进程的线性地址空间内。再如，两个进程操作这个页面，一个读一个写，也不能产生页写保护异常把页面另复制一份，否则无法共享管道。下面我们来看管道的具体创建过程。

1.为管道文件在file_table[64]中申请空闲项

创建文件都是让当前进程（一个进程）使用，而管道文件就是为了两个进程（读管道进程和写管道进程）的使用而创建的。实例1中管道是由父进程（写管道进程）创建的。父进程在创建管道时，处处为子进程（读管道进程）做准备，

使得子进程一旦被创建，天然就具备操作管道的能力。

父进程先在file_table[64]中申请“两个”空闲项，并将这两个空闲项的引用计数设置为1，表示它们被引用了，父子进程以后操作管道文件可以各用一项。执行代码如下：

```
//代码路径: fs/pipe.c:

int sys_pipe (unsigned long * fildes)

{

    struct m_inode * inode;

    struct file * f[2];

    int fd[2];

    int i,j;

    j=0;
```

```
    for (i=0; j<2&& i<NR_FILE; i++) //准备在file_table[64]中申  
    请两个空闲项
```

```
    if (! file_table[i].f_count) //找到空闲项
```

```
        (f[j++]=i+file_table) -> f_count++; //每项引用计数为1
```

```
    if (j==1)
```

```
        f[0]-> f_count=0;
```

```
    if (j<2)
```

```
        return-1;
```

```
    .....
```

```
    }
```

为创建管道文件而在file_table[64]中申请的两个空闲项如图8-2所示。

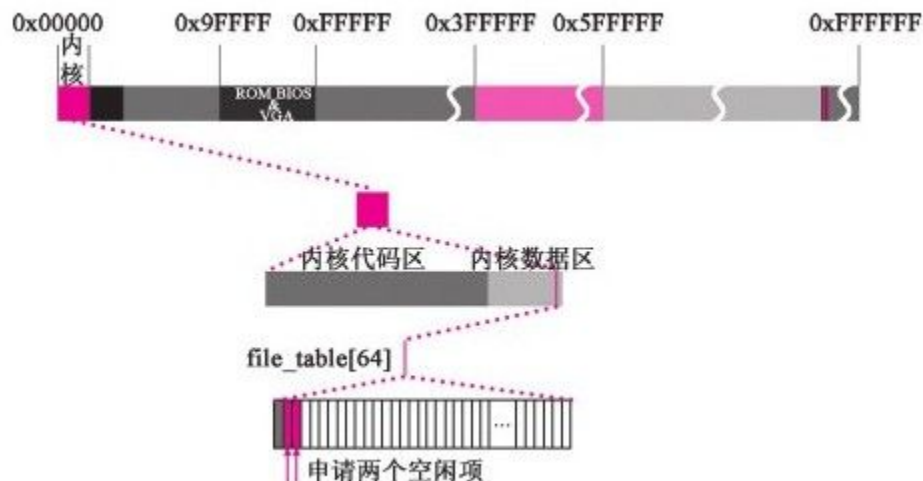


图 8-2 为创建管道文件而在file_table[64]中申请两个空闲项

2.进程task_struct中的*filp[20]与file_table[64]中的表项挂接

父进程task_struct在*filp[20]中申请两个空闲项，分别与前面在file_table[64]中申请的两个表项相挂接。这样，当前进程文件结构*filp[20]中就有两个表项与file_table[64]建立关系。等到它作为父进程创建子进程时，*filp[20]中的这两个表项就会

自然而然地复制给它的子进程，使之也“天然地”和file_table[64]结构中同样的管道文件表项建立了关系，执行代码如下：

```
//代码路径: fs/pipe.c:

int sys_pipe (unsigned long * fildes)
{
    .....

    if (j==1)

        f[0]->f_count=0;

    if (j<2)

        return-1;

    j=0;

    for (i=0; j<2&& i<NR_OPEN; i++) //准备在*filp[20]中申请
    两个空闲项

        if (! current->filp[i]) { //找到空闲项

            current->filp[fd[j]=i]=f[j]; //分别与file_table[64]中申请的两个空
            闲项挂接
```

```
j++;  
  
}  
  
if (j==1)  
  
current->filp[fd[0]]=NULL;  
  
if (j<2) {  
  
f[0]->f_count=f[1]->f_count=0;  
  
return-1;  
  
}  
  
.....  
  
}
```

图8-3表示了将当前进程与管道文件建立关联的效果。

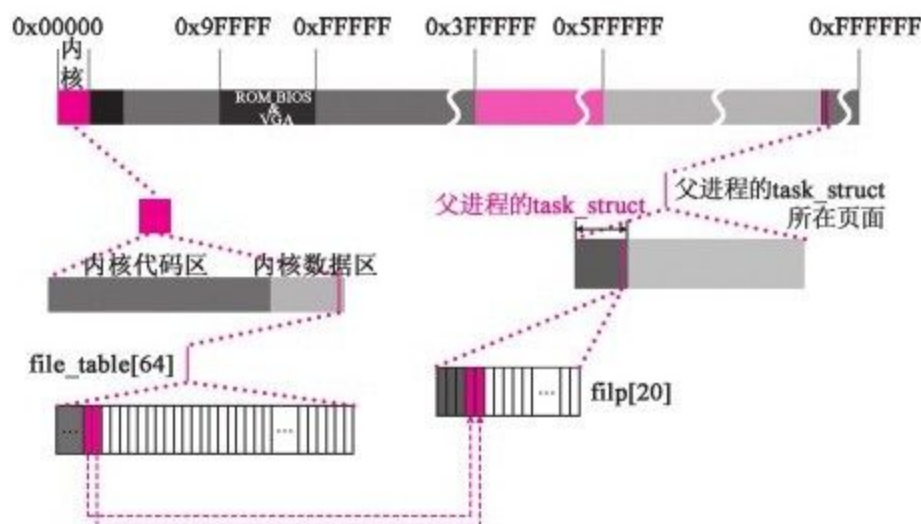


图 8-3 将当前进程与管道文件建立联系

3.创建管道文件i节点

进程要想具备操作管道文件的能力，还要建立管道文件i节点与file_table[64]的关系。为此调用get_pipe_inode () 函数，先为该管道文件在inode_table[32]中申请一个i节点。执行代码如下：

//代码路径： fs/pipe.c:

```

int sys_pipe (unsigned long * fildes)

{

.....

if (j==1)

current->filp[fd[0]]=NULL;

if (j<2) {

f[0]->f_count=f[1]->f_count=0;

return-1;

}

if (! (inode=get_pipe_inode () ) ) { //创建管道文件i节点

current->filp[fd[0]]=

current->filp[fd[1]]=NULL;

f[0]->f_count=f[1]->f_count=0;

return-1;

}

.....

}

```

由于管道的本质就是一个内存页面，系统申请一个空闲内存页面，并将该页面的地址载入i节点。值得注意的是，此刻inode->i_size字段承载的不再是文件大小，而是内存页面的起始地址。执行代码如下：

```
//代码路径: fs/inode.c:

struct m_inode * get_pipe_inode (void)

{

struct m_inode * inode;

if (! (inode=get_empty_inode () ) )

return NULL;

if (! (inode->i_size=get_free_page () ) ) { //申请页面作为管道

inode->i_count=0;

return NULL;
```

```
}  
  
inode->i_count=2; /*sum of readers/writers*/  
  
.....  
  
}
```

管道文件也是文件，所以也要有i节点。有i节点就要有引用计数。在Linux 0.11中，默认操作这个管道文件的进程“能且仅能”有两个，一个是读进程，另一个是写进程，所以这里直接设置为2。

之后，让读管道指针和写管道指针都指向管道（其实就是这个空闲页面）的起始位置，以便将来读写管道的进程操作，并将该i节点的属性设置为“管道型i节点”，以此来标识该i节点的特殊性，即它并不是实际存储在硬盘上的文件的i节点，只是一个内存页面。执行代码如下：

//代码路径: fs/inode.c:

```
struct m_inode * get_pipe_inode (void)
```

```
{
```

```
.....
```

```
if (! (inode->i_size=get_free_page () ) ) {
```

```
inode->i_count=0;
```

```
return NULL;
```

```
}
```

```
inode->i_count=2; /*sum of readers/writers*///引用计数设置为2
```

PIPE_HEAD (*inode) =PIPE_TAIL (*inode) =0; //PIPE_HEAD
为写管道指针, PIPE_TAIL为读管道指针, 都设置为0

```
inode->i_pipe=1; //设置管道文件属性
```

```
return inode;
```

```
}
```

为管道文件申请i节点并对其进行设置的过程
如图8-4所示。

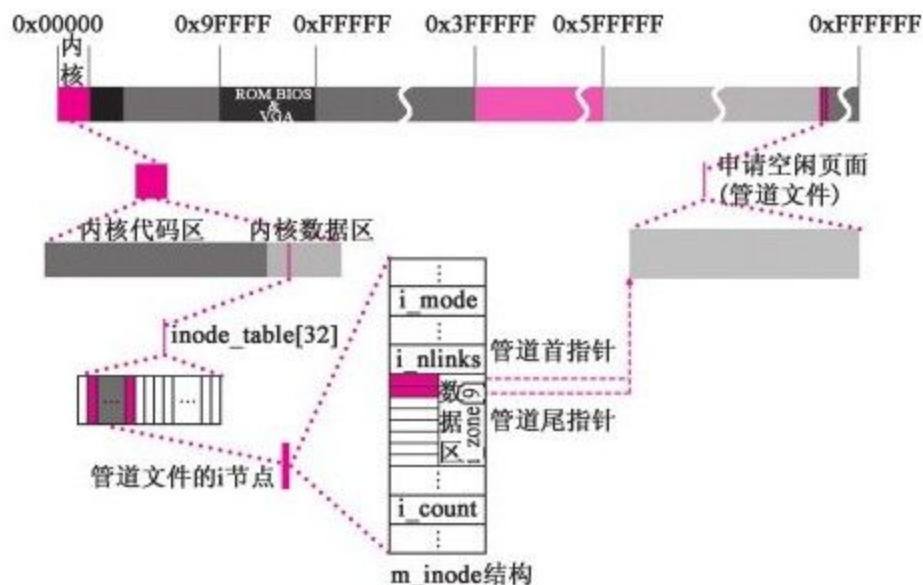


图 8-4 为管道文件创建i节点

4.将管道文件i节点与file_table[64]建立联系

管道文件i节点已经设置完毕，现在就可以将它与file_table[64]建立关系了，具体的操作是，始化file_table[64]中的两个空闲项，让它们都指向这个管道文件i节点，文件读写指针都指向管道的起始位置。第1个空闲项的文件模式置为读，第2个空闲项的文件模式置为写，这样，父进程已经具

备了操作管道文件的能力，由它创建的子进程也将天然具备操作管道文件的能力，执行代码如下：

```
//代码路径: fs/pipe.c:

int sys_pipe (unsigned long * fildes)

{

.....

if ( ! (inode=get_pipe_inode ( ) ) ) {

current->filp[fd[0]]=

current->filp[fd[1]]=NULL;

f[0]->f_count=f[1]->f_count=0;

return-1;

}

f[0]->f_inode=f[1]->f_inode=inode; //i节点和表项挂接

f[0]->f_pos=f[1]->f_pos=0; //文件指针归0
```

```

f[0]->f_mode=1; /*read*///设置为读模式

f[1]->f_mode=2; /*write*///设置为写模式

put_fs_long (fd[0], 0+fildes) ;

put_fs_long (fd[1], 1+fildes) ;

return 0;

}

```

将管道文件i节点与file_table[64]建立联系的结果如图8-5所示。

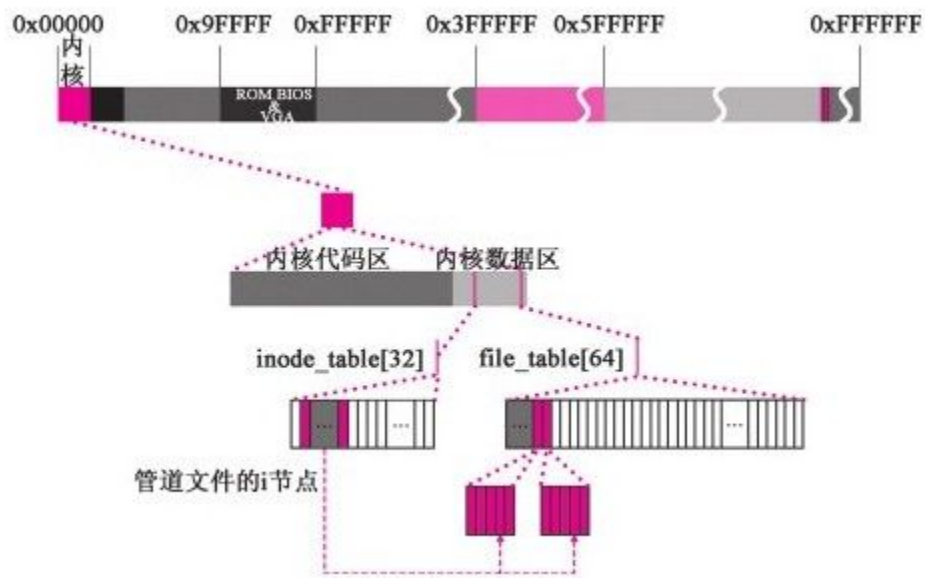


图 8-5 管道文件i节点与file_table[64]建立联系

5.将管道文件句柄返给用户进程

现在将管道文件的两个句柄返给用户进程，即返给实例1代码中的fd[2]。这个数组有两项，每一项分别存放一个句柄，这样子进程也将继承这两个文件句柄，父子两个进程就可以通过不同的文件句柄操作这个管道文件了。执行代码如下：

//代码路径: fs/pipe.c:

```
int sys_pipe (unsigned long * fildes)
```

```
{
```

```
.....
```

```
f[0]->f_inode=f[1]->f_inode=inode;
```

```
f[0]->f_pos=f[1]->f_pos=0;
```

```
f[0]->f_mode=1; /*read*/
```

```
f[1]->f_mode=2; /*write*/
```

```
put_fs_long (fd[0], 0+fildes) ; //设置读管道文件句柄
```

```
put_fs_long (fd[1], 1+fildes) ; //设置写管道文件句柄

return 0;

}
```

将管道文件句柄返给用户进程的结果如图8-6所示。

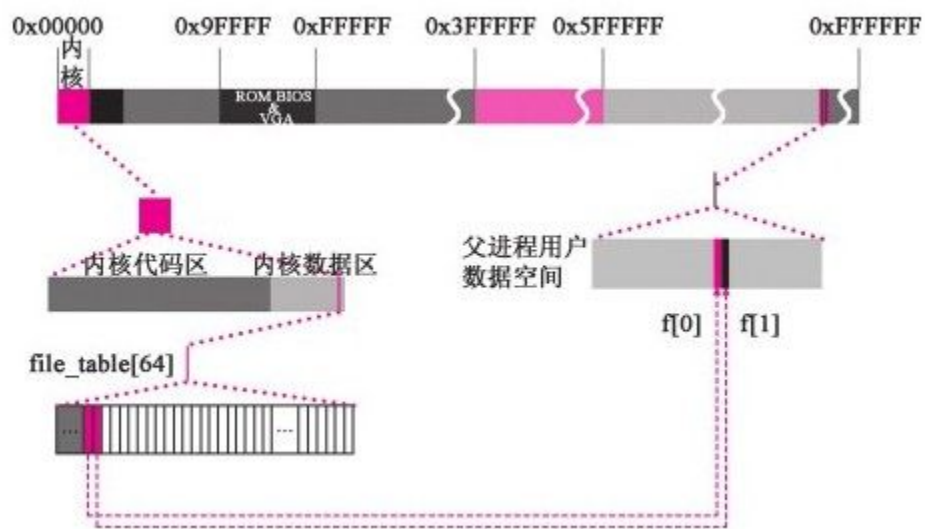


图 8-6 将管道文件句柄返给用户进程

8.1.2 管道的操作

Linux 0.11管道操作要实现的效果是，读管道进程执行时，如果管道中有未读数据，就读取数据，没有未读数据，就挂起，这样就不会读取垃圾数据；写管道进程执行时，如果管道中有剩余空间，就写入数据，没有剩余空间了，就挂起，这样就不会覆盖尚未读取的数据。另外，管道大小只有一个页面，所以写或读到页面尾端后，读写指针要能够回滚到页面首端以便继续操作。

回滚的实现代码如下：

```
//代码路径：fs/read_write.c:
```

```
int sys_read (unsigned int fd,char * buf,int count) //读管道指针
```

```
{
```

```

.....

while (count > 0) {

.....

if (chars > size)

chars=size;

count-=chars;

read+=chars;

size=PIPE_TAIL (*inode) ;

PIPE_TAIL (*inode) +=chars; //读多少数据，指针就偏移多少

PIPE_TAIL (*inode) &= (PAGE_SIZE-1) ; //指针超过一个页面，
(&=) 操作可以实现自动回滚

while (chars-->0)

put_fs_byte ( ( (char *) inode->i_size) [size++], buf++) ;

}

.....

}

int write_pipe (struct m_inode * inode,char * buf,int count) //写管道
指针

```

```

{

.....

while (count > 0) {

.....

if (chars > size)

chars=size;

count-=chars;

written+=chars;

size=PIPE_HEAD (*inode) ;

PIPE_HEAD (*inode) +=chars; //写多少数据，指针就偏移多少

PIPE_HEAD (*inode) &= (PAGE_SIZE-1) ; //指针超过一个页面，
(&=) 操作可以实现自动回滚

while (chars-->0)

    ((char *) inode->i_size) [size++]=get_fs_byte (buf++) ;

}

.....

}

```

在不断回滚操作的情况下，控制写入和读取，以及将进程唤醒和挂起的代码如下：

```
//代码路径: include/linux/fs.h:

.....

#define PIPE_HEAD (inode) ( (inode) .i_zone[0])

#define PIPE_TAIL (inode) ( (inode) .i_zone[1])

#define PIPE_SIZE (inode) ( (PIPE_HEAD (inode) -
PIPE_TAIL (inode) ) & (PAGE_SIZE-1) )

.....

//代码路径: fs/read_write.c:

int sys_read (unsigned int fd,char * buf,int count) //读管道指针

{

.....

while (count > 0) {

    while (! (size=PIPE_SIZE (*inode) ) ) { //读写指针重合时,
就视为把管道中数据都读完了
```

wake_up (&inode->i_wait) ; //管道数据都读完了，唤醒写管道进程

if (inode->i_count != 2) /*are there any writers?*/

return read;

sleep_on (&inode->i_wait) ; //没数据读了，将读管道进程挂起

}

chars=PAGE_SIZE-PIPE_TAIL (*inode) ;

if (chars > count)

chars=count;

if (chars > size)

chars=size;

.....

}

wake_up (&inode->i_wait) ; //读取了数据，意味着管道有剩余空间了，唤醒写管道进程

return read;

}

int write_pipe (struct m_inode * inode,char * buf,int count) //写管道指针{

.....

```
while (count > 0) {
```

```
    while (! (size= (PAGE_SIZE-1) -PIPE_SIZE (*inode) ) ) {  
        写指针最多能写入4095 t字节的数据，就视为把管道写满了（一个页面  
        大小为4096字节）
```

```
        wake_up (&inode->i_wait) ; //管道写满了，有数据了，唤醒读  
        管道进程
```

```
        if (inode->i_count != 2) { /*no readers*/
```

```
            current->signal|= (1 << (SIGPIPE-1) ) ;
```

```
            return written?written: -1;
```

```
        }
```

```
        sleep_on (&inode->i_wait) ; //没剩余空间了，将写管道进程挂  
        起
```

```
    }
```

```
    chars=PAGE_SIZE-PIPE_HEAD (*inode) ;
```

```
    if (chars > count)
```

```
        chars=count;
```

```
    if (chars > size)
```

```
        chars=size;
```

```
.....  
  
}  
  
wake_up (&inode->i_wait) ; //写入了数据，意味着管道有数据  
了，唤醒读管道进程  
  
return written;  
  
}
```

当管道中所有可写空间全被写满时，写管道指针回滚一圈，与读管道指针差1字节，这时应把写管道进程挂起。Linux 0.11将sys_write（）函数设计为，写管道进程一次最多只能写4095字节。

下面我们通过实例1，介绍管道操作的过程。

1.读管道进程开始操作管道文件

实例1中父进程创建完管道后，开始创建子进程，即读管道进程。创建完毕后，我们不妨假设

此时系统中只有读管道和写管道两个进程处于就绪态，而且读管道进程先执行，执行实例1中“`read (fd[0], str2, strlen (str2))`”这行源代码。`read ()` 函数会映射到系统调用函数`sys_read ()` 中去执行，并最终执行到`read_pipe ()` 函数中。由于此时管道内没有任何数据，所以此时系统会将读管道进程挂起，然后切换到写管道进程中去执行。执行代码如下：

```
//代码路径: fs/read_write.c:

int sys_read (unsigned int fd,char * buf,int count)

{

.....

verify_area (buf,count) ;

inode=file->f_inode;

if (inode->i_pipe)
```



```
    return (file->f_mode&1) ?read_pipe (inode,buf,count) : -  
EIO; //调用读管道函数
```

```
    if (S_ISCHR (inode->i_mode) )
```

```
        return rw_char (READ,inode->i_zone[0], buf,count, &file->  
f_pos) ;
```

```
        .....
```

```
    }
```

```
    //代码路径: fs/pipe.c:
```

```
int read_pipe (struct m_inode * inode,char * buf,int count)
```

```
//读管道函数
```

```
{
```

```
    int chars,size,read=0;
```

```
    while (count>0) {
```

```
        while ( ! (size=PIPE_SIZE (*inode) ) ) { //管道中没有数据,  
        进入此循环执行
```

```
            wake_up (&inode->i_wait) ;
```

```
            if (inode->i_count !=2) /*are there any writers?*/
```

```
            return read;
```

sleep_on (&inode->i_wait); //将读管道进程挂起，切换到写管道进程（已经假设系统中只有操作管道的两个进程处于就绪态）

}

chars=PAGE_SIZE-PIPE_TAIL (*inode) ;

if (chars > count)

chars=count;

.....

}

.....

}

此时管道内没有任何数据，读管道进程被挂起，进程的状态如图8-7所示。

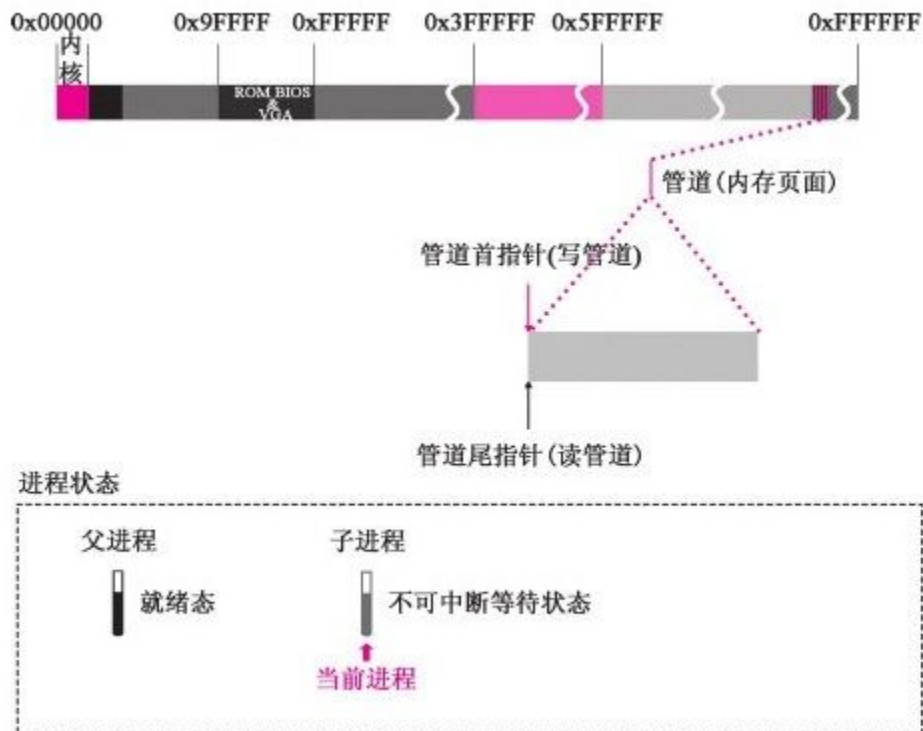


图 8-7 进程开始操作管道——挂起读进程

2.写管道进程向管道中写入数据

写管道进程开始执行，它会将实例1中的str1数组中指定的1024字节的数据循环地写入管道内，即执行“write (fd[1], str1, strlen

(str1))”这行源代码。write () 函数会映射到系统调用函数sys_write () 中去执行，并最终执

行到write_pipe () 函数中。写完后，管道中就已经有数据可以读出，唤醒读管道进程（唤醒了读管道进程并不等于读管道进程就立即执行），此次写管道操作就执行完毕。执行代码如下：

```
//代码路径: fs/read_write.c:

int sys_write (unsigned int fd,char * buf,int count)

{

.....

if (! count)

return 0;

inode=file->f_inode;

if (inode->i_pipe)

return (file->f_mode&2) ?write_pipe (inode,buf,count) : -
EIO; //调用写管道函数

if (S_ISCHR (inode->i_mode) )
```

```
return rw_char (WRITE,inode->i_zone[0], buf,count, &file->f_pos) ;
```

```
.....
```

```
}
```

//代码路径: fs/pipe.c:

int write_pipe (struct m_inode * inode,char * buf,int count) //写管道函数

```
{
```

```
int chars,size,written=0;
```

```
while (count>0) {
```

```
.....
```

```
size=PIPE_HEAD (*inode) ;
```

```
PIPE_HEAD (*inode) +=chars;
```

```
PIPE_HEAD (*inode) &= (PAGE_SIZE-1) ;
```

```
while (chars-->0)
```

```
( (char *) inode->i_size) [size++]=get_fs_byte (buf++) ; //向管道中写入数据
```

```
}
```

```
wake_up (&inode->i_wait) ; //唤醒读管道进程
```

}

写管道进程向管道中写入数据的过程如图8-8所示。

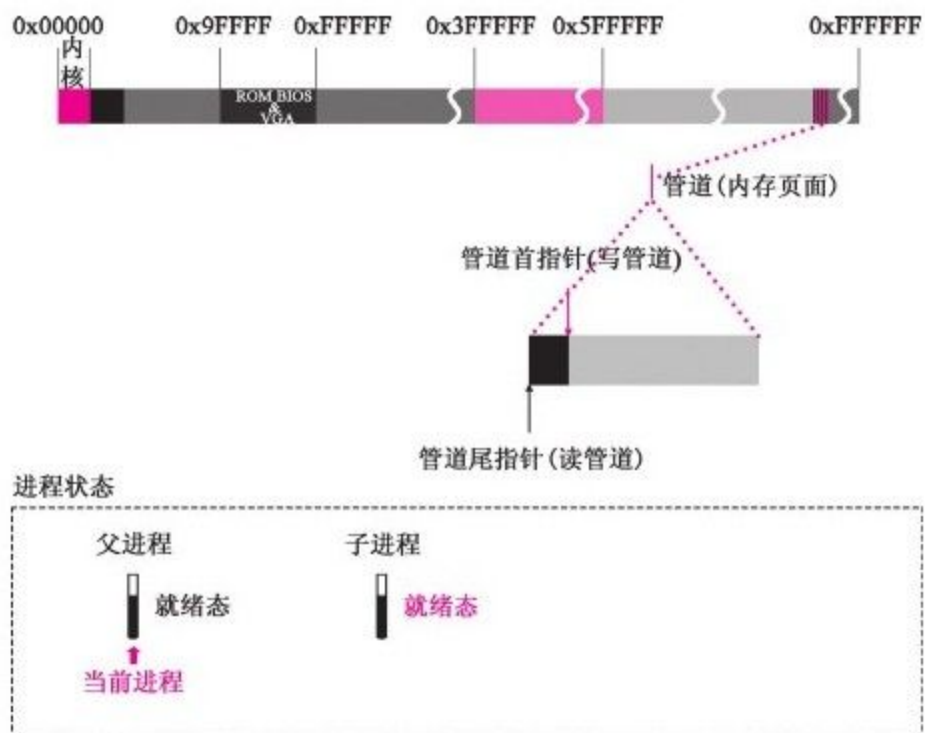


图 8-8 写管道进程向管道中写入数据

3.写管道进程继续向管道写入数据

当前进程是写管道进程，写完一次管道之后将返回用户空间。通过实例1中“for (i=0; i<10000; i++)”这行代码我们得知，写管道要操作10 000次，而写管道进程的时间片还没有用完，它还要继续执行写管道操作。

向管道中不断写入数据的过程如图8-9所示。



图 8-9 写管道进程不断向管道中写入数据

4.写管道进程已将管道空间写满

不妨假设在写管道进程工作的过程中，发生了时钟中断，削减了它的时间片，只要时间片不被削减为0，它就会继续执行。对应代码如下：

//代码路径：kernel/sched.c:

```
void do_timer (long cpl) //时钟中断处理函数
```

```
{
```

```
.....
```

```
if (next_timer) {
```

```
next_timer->jiffies--;
```

```
while (next_timer && next_timer->jiffies <= 0) {
```

```
void (*fn) (void) ;
```

```
fn=next_timer->fn;
```

```
next_timer->fn=NULL;
```

```
next_timer=next_timer->next;
```



```

    (fn)    () ;

}

}

if (current_DOR&0xf0)

do_floppy_timer () ;

if ( (--current->counter) > 0) return; //时间片不为0, 直接返回

current->counter=0;

if (! cpl) return;

schedule () ;

}

```

图8-10的下方表示了发生时钟中断后对写管道进程的影响。

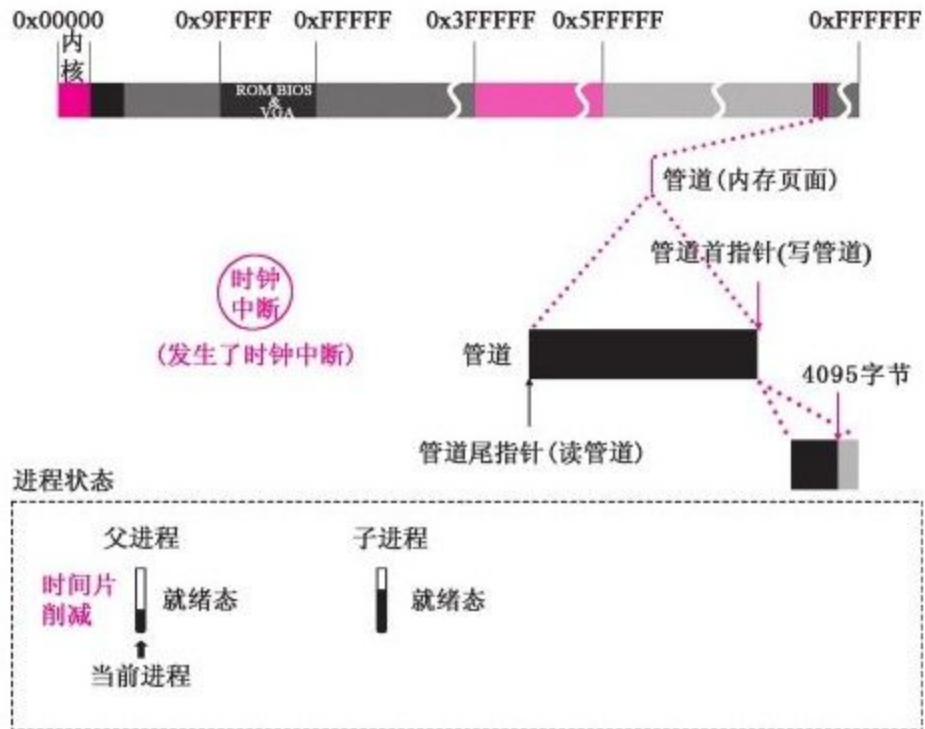


图 8-10 写管道执行过程中产生时钟中断

直到写管道进程把管道写满为止（写入4095字节就算满了）。在写入的过程中，写管道指针一直指向数据的写入位置，一直向管道尾端移动。

5.写管道进程挂起

写满后，系统就要将写管道进程挂起，然后切换到读管道进程去执行。执行代码如下：

//代码路径：fs/pipe.c:

```
int write_pipe (struct m_inode * inode,char * buf,int count)

{

int chars,size,written=0;

while (count>0) {

    while ( ! (size= (PAGE_SIZE-1) -PIPE_SIZE (*inode) ) ) { //
写4095字节就算满了，此条件为真，进入while执行

    wake_up (&inode->i_wait) ;

    if (inode->i_count !=2) { /*no readers*/

    current->signal|= (1<< (SIGPIPE-1) ) ;

    return written?written: -1;

    }

    sleep_on (&inode->i_wait) ; //写管道进程挂起，切换到读管道
进程去执行

    }
```

.....

}

.....

}

写管道进程挂起并切换到读管道进程的过程
如图8-11所示。

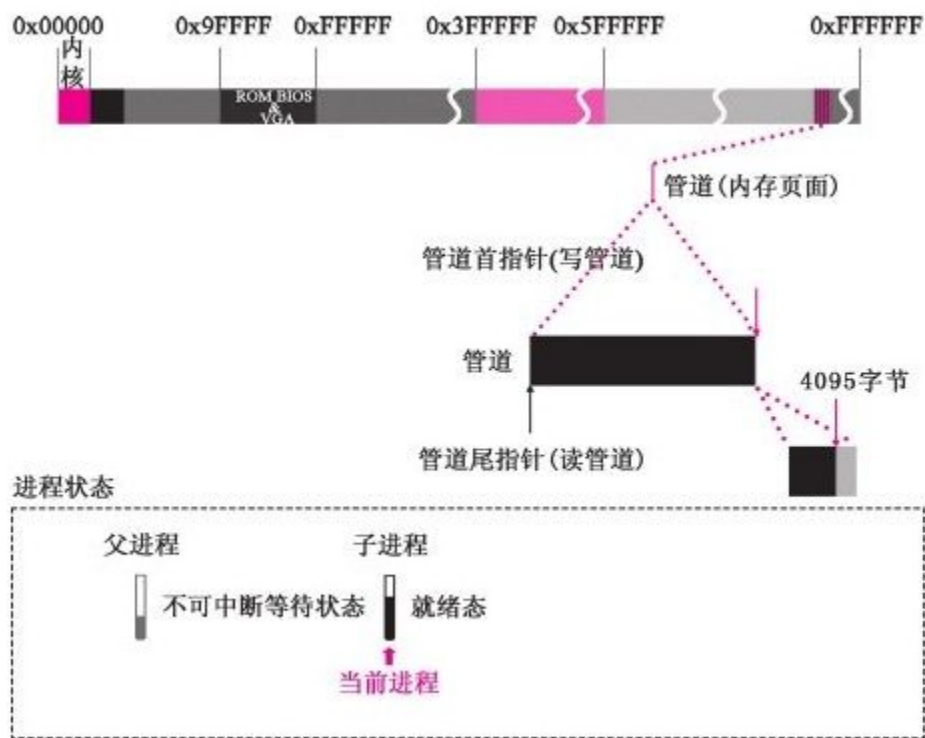


图 8-11 写管道进程挂起并唤醒读管道进程

6.读管道进程从管道中读出数据

读管道进程将继续在`read_pipe ()` 函数中执行。根据实例1的代码，此次执行将会把管道中512字节的数据读入读管道进程的用户空间内。执行代码如下：

//代码路径: fs/pipe.c:

```
int read_pipe (struct m_inode * inode,char * buf,int count)
{
.....

while (count>0) {//读取512字节的数据

.....

chars=PAGE_SIZE-PIPE_TAIL (*inode) ;

if (chars>count)

chars=count;

if (chars>size)
```

```

chars=size;

count-=chars;

read+=chars;

size=PIPE_TAIL (*inode) ;

PIPE_TAIL (*inode) +=chars; //读多少，指针移动多少

PIPE_TAIL (*inode) &= (PAGE_SIZE-1) ;

while (chars-->0)

    put_fs_byte ( ( (char *) inode->i_size) [size++], buf++) ; //
读数据的执行代码

}

.....

}

```

读出了数据，意味着有了剩余空间，系统此时会唤醒写管道进程。执行代码如下：

//代码路径：fs/pipe.c:

```

int read_pipe (struct m_inode * inode,char * buf,int count)

{

.....

while (count>0) {

.....

size=PIPE_TAIL (*inode) ;

PIPE_TAIL (*inode) +=chars;

PIPE_TAIL (*inode) &= (PAGE_SIZE-1) ;

while (chars-->0)

put_fs_byte ( ( (char *) inode->i_size) [size++], buf++) ;

}

wake_up (&inode->i_wait) ; //唤醒写管道进程

}

```

读管道进程从管道中读出数据的过程如图8-12所示。

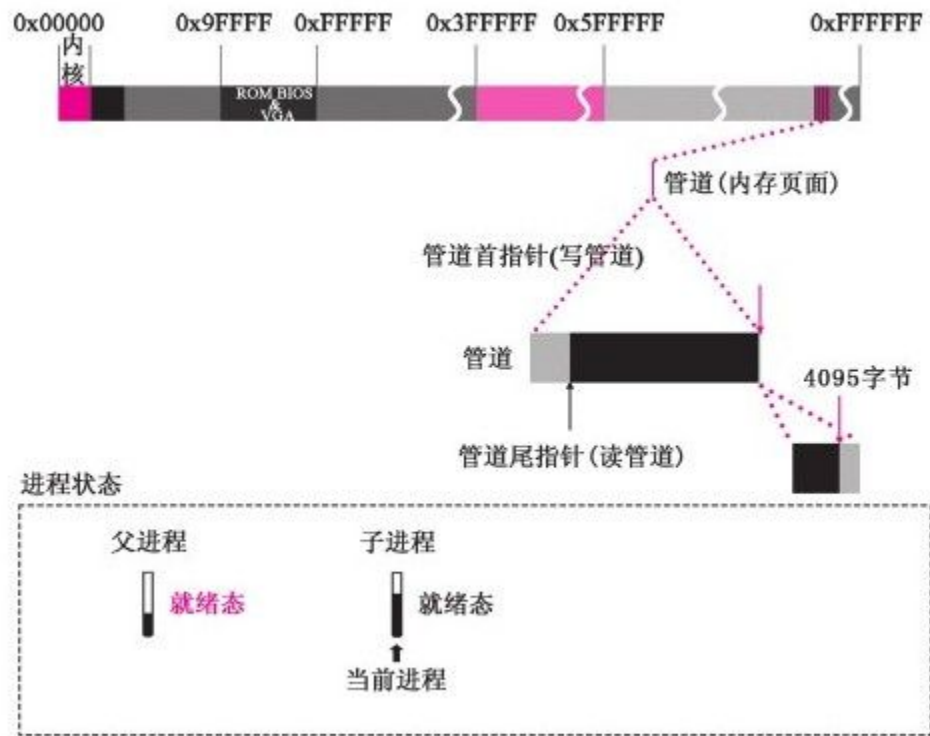


图 8-12 读管道进程从管道中读出数据

7.读管道进程继续执行，不断从管道中读出数据

当前进程还是读管道进程，读一次管道之后将返回用户空间。通过实例1中“for (j=0; j<20000; j++)”这行代码我们得知，读管道要操作20 000次，而且读管道进程的时间片还没有用

完，所以还要继续执行读管道操作。读管道进程不断从管道中读出数据的过程如图8-13所示。

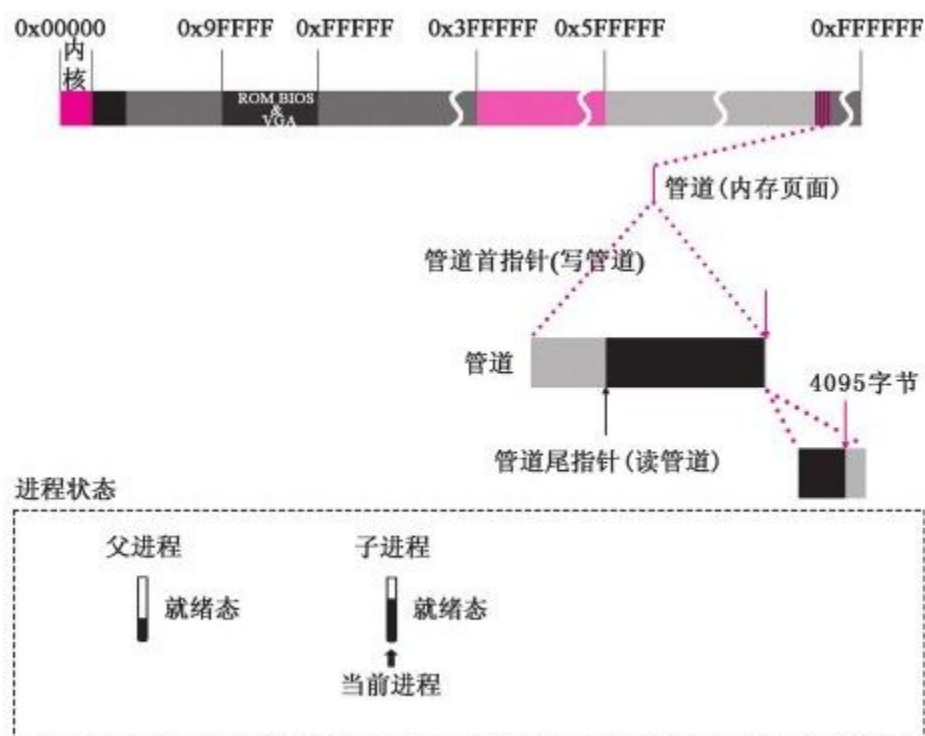


图 8-13 读管道进程不断从管道读出数据

8.读管道进程执行中发生时钟中断

假设在读管道进程工作的过程中，也发生时钟中断，削减了它的时间片，只要不削减为0，它

就继续执行。执行代码如下：

```
//代码路径: kernel/schde.c:

void do_timer (long cpl)

{

.....

if (next_timer) {

next_timer->jiffies--;

while (next_timer&&next_timer->jiffies<=0) {

void (*fn) (void) ;

fn=next_timer->fn;

next_timer->fn=NULL;

next_timer=next_timer->next;

(fn) () ;

}

}

if (current_DOR&0xf0)
```

```
do_floppy_timer ( ) ;  
  
if ( (--current->counter) > 0) return; //时间片不为0, 直接返回  
  
current->counter=0;  
  
if ( ! cpl) return;  
  
schedule ( ) ;  
  
}
```

读管道进程执行过程中发生时钟中断的处理方法如图8-14所示。注意图中读管道进程的进程条，它的时间片已经削减。

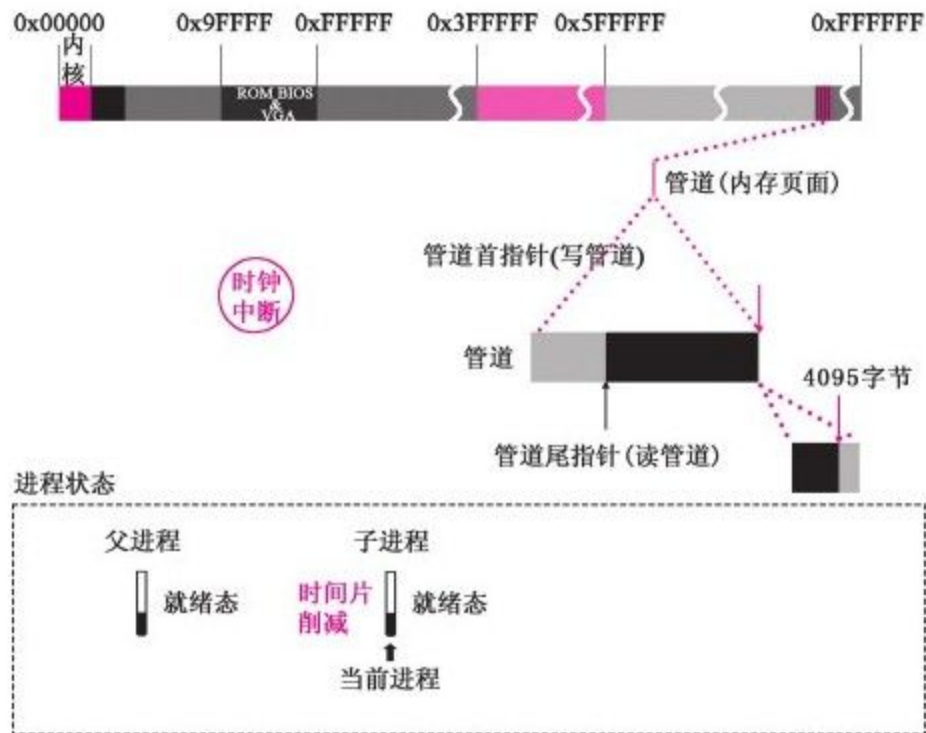


图 8-14 读管道进程执行过程中发生时钟中断

9.读管道进程执行过程中再次发生时钟中断

读管道进程执行过程中又一次发生时钟中断后，读管道进程时间片为0，它被挂起并切换到写管道进程去执行。执行代码如下：

//代码路径：kernel/schde.c:

```
void do_timer (long cpl)

{

.....

if (current_DOR&0xf0)

do_floppy_timer ();

if ( (--current->counter) > 0) return; //时间片为0

current->counter=0;

if (! cpl) return;

schedule (); //进程切换

}
```

对于时钟中断的处理如图8-15所示。图中代表读管道进程的进程条的时间片已经削减为0。

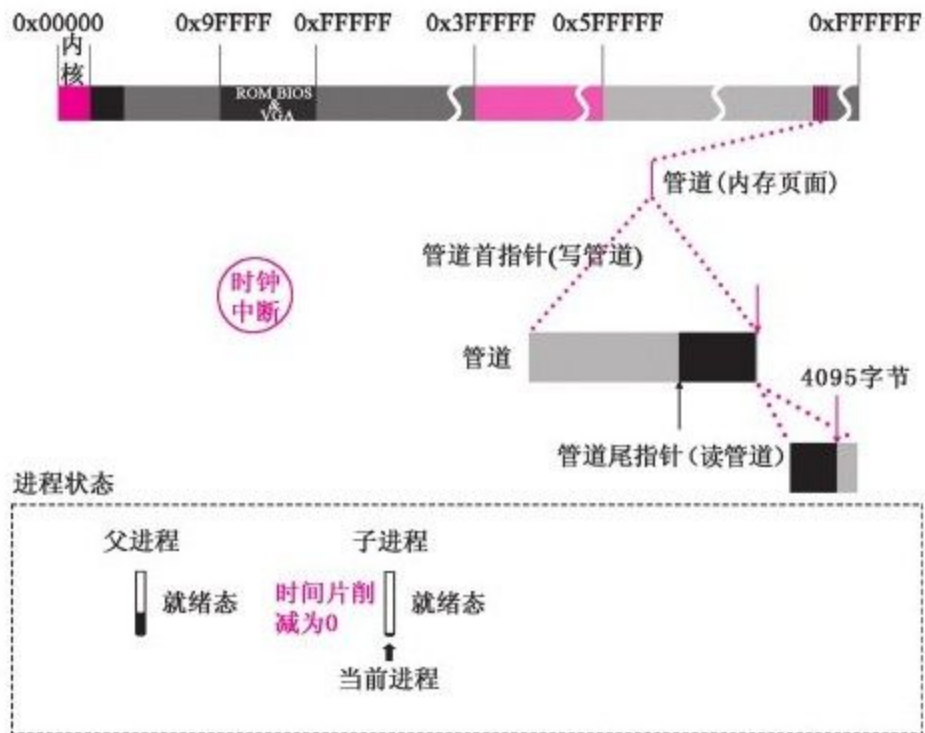


图 8-15 读进程执行过程中发生时钟中断

值得注意的是，两次时钟中断的产生并不会影响到数据的写入或读出，根本原因是数据的写入和读出都是在0特权级的内核代码中进行的，完全由系统控制，只会削减时间片，不会影响数据操作的执行。

10.读管道进程切换到写管道进程执行

写管道进程挂起前，管道操作指针已经被移动至管道首端。接下来。写管道进程将从管道首端开始，继续往管道中写入数据，直到再次没有剩余空间为止。该过程如图8-16所示。

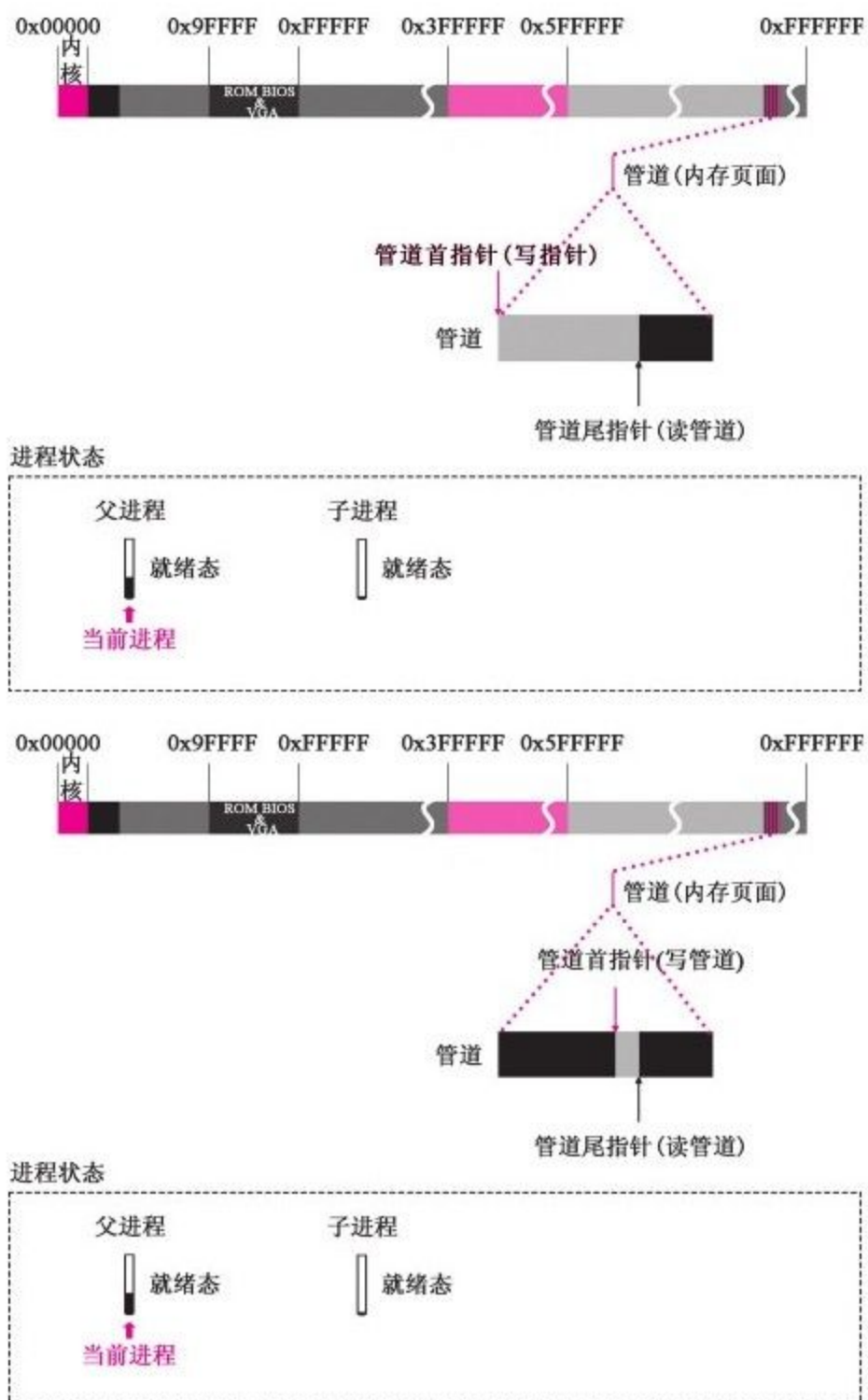


图 8-16 写管道进程继续写入数据

11.写管道进程挂起，切换到读管道进程执行

管道中再次被写满后，写管道进程又要被系统挂起，之后就切换到读管道进程去执行。Linux 0.11重新分配时间片的原则是当所有处于就绪态的进程时间片均为0时，分配时间片。由于此时读管道进程是唯一处于就绪态的进程，并且它的时间片也用完了，重新给它们分配时间片，执行代码如下：

//代码路径：kernel/schde.c:

```
void schedule (void)
```

```
{
```

```
.....
```

```
while (1) {
```

```
    c=-1;
```

```

next=0;

i=NR_TASKS;

p=&task[NR_TASKS];

while (--i) {

if (! *--p)

continue;

if ( (*p) -> state==TASK_RUNNING&& (*p) -> counter > c)

c= (*p) -> counter,next=i;

}

if (c) break;

for (p=&LAST_TASK; p> &FIRST_TASK; --p) //分配时间片

if (*p)

    (*p) -> counter= ( (*p) -> counter>>1) +

    (*p) -> priority;

}

switch _to (next) ;

}

```

然后，读管道进程继续执行，读取一部分数据的过程如图8-17所示。

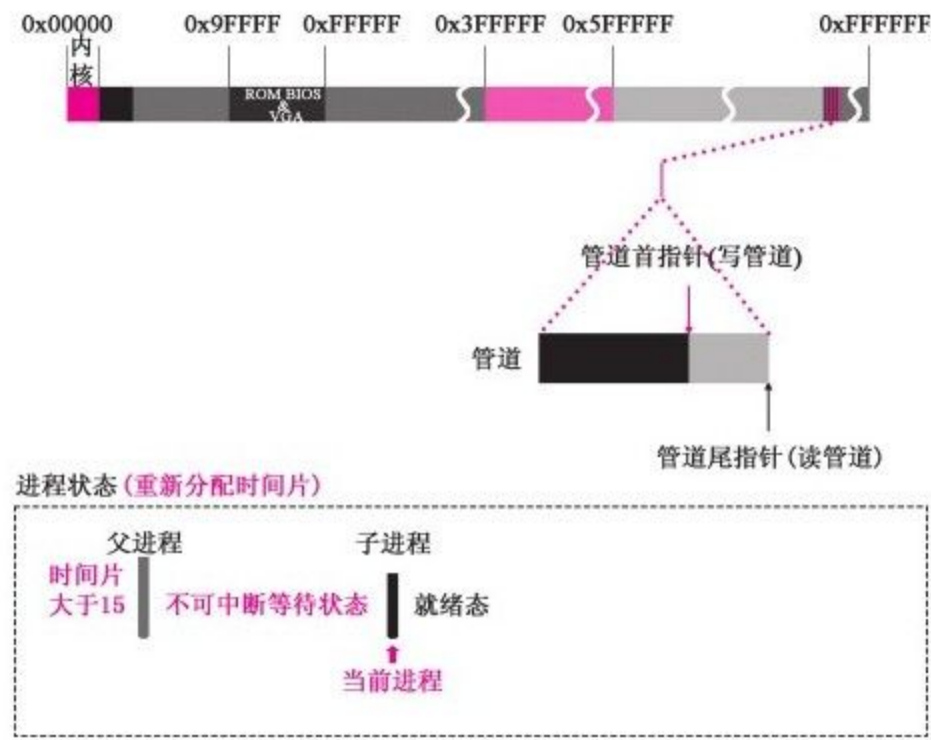


图 8-17 读管道进程继续从管道中读取数据

12.读管道进程继续执行，直到把管道中的数据读完

读管道进程开始执行后，将继续把管道中的数据读出。当操作到管道尾端后，也会将读管道指针从管道尾端移动至管道首端，并从首端继续读取管道中的内容，直至将数据彻底读完，两个指针重合。执行代码如下：

```
//代码路径: fs/pipe.c:
```

```
int read_pipe (struct m_inode * inode,char * buf,int count)
```

```
{
```

```
.....
```

```
while (count > 0) {
```

```
    while (! (size=PIPE_SIZE (*inode) )) { //指针重合，证明数据读完了
```

```
        wake_up (&inode->i_wait) ; //唤醒写管道进程
```

```
        if (inode->i_count != 2) /*are there any writers?*/
```

```
            return read;
```

sleep_on (&inode->i_wait) ; //将读管道进程挂起，切换到写管道进程执行

}

chars=PAGE_SIZE-PIPE_TAIL (*inode) ;

if (chars > count)

chars=count;

if (chars > size)

chars=size;

count-=chars;

read+=chars;

size=PIPE_TAIL (*inode) ;

PIPE_TAIL (*inode) +=chars;

PIPE_TAIL (*inode) &= (PAGE_SIZE-1) ;

while (chars-->0)

put_fs_byte (((char *) inode->i_size) [size++], buf++) ; //

读取数据

}

.....

}

该过程如图8-18所示。

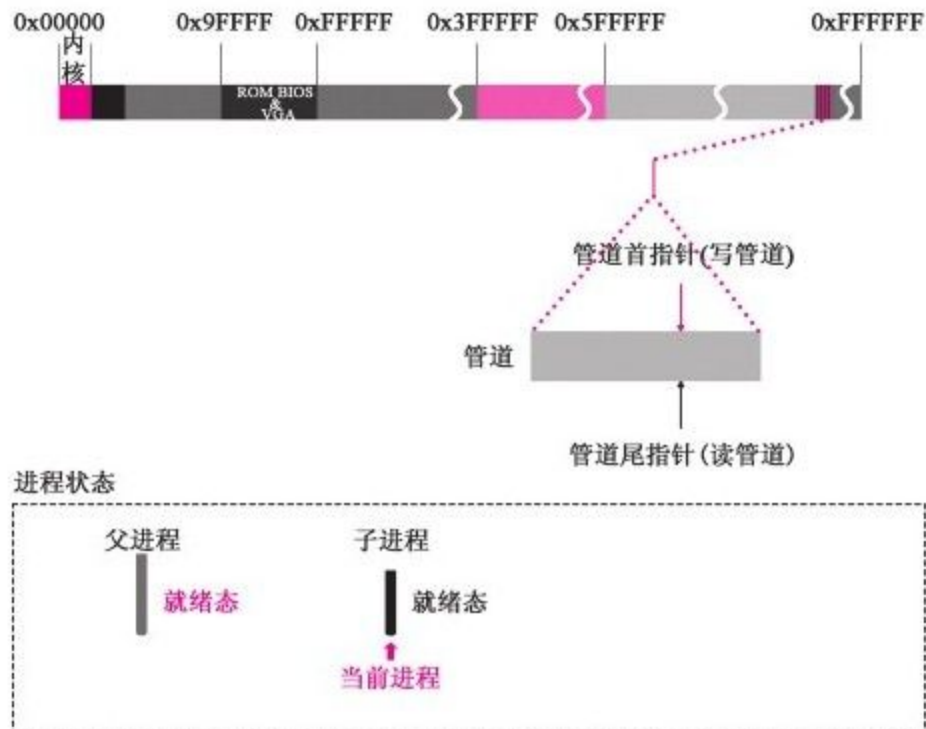


图 8-18 读管道进程将管道中的数据读完

从前面对管道操作的介绍中不难发现，两个进程占用一个管道的标志是，在file_table[64]中两

个进程分别占据一个管道文件的file项，就可以对管道进行操作。

如果进程A创建两个管道，并创建进程B,A、B两个进程就可以利用这两个管道进行逆向的数据交互，情景如图8-19所示。

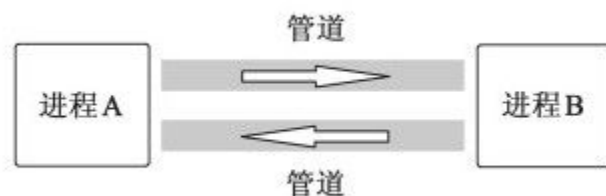


图 8-19 两个进程使用管道逆向数据交互

如果进程A创建六个管道，并创建进程B和进程C,A、B、C三个进程就可以利用这六个管道进行两两逆向的数据交互，情景如图8-20所示。

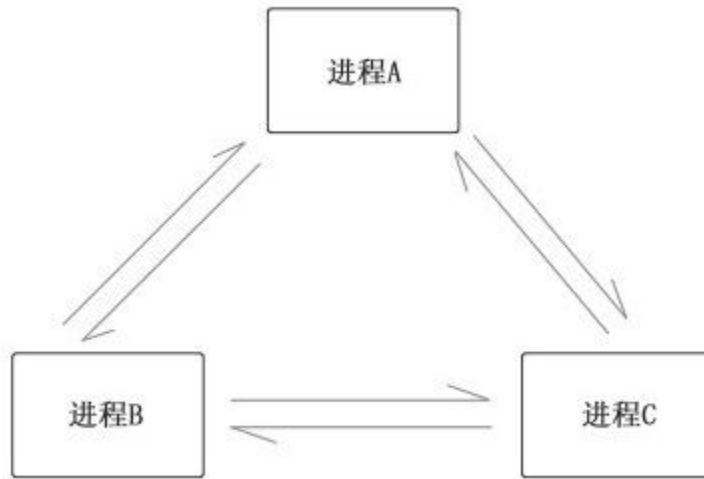


图 8-20 三个进程使用管道两两逆向数据交互

只要进程的总量不超过64、进程占据的file项的数量不超过file_table[64]的承载力，就可以构建出任意复杂的管道组合操作结构。

8.2 信号机制

信号机制是Linux 0.11为进程提供的一套“局部的类中断机制”，即在进程执行的过程中，如果系统发现某个进程接收到了信号，就暂时打断进程的执行，转而去执行该进程的信号处理程序，处理完毕后，再从进程“被打断”之处继续执行。

本节将分两部分对信号机制进行详细的介绍。

第一部分：通过实例2的执行过程，对系统以及进程处理信号的过程进行详细的介绍。

第二部分：系统通过对信号的分析来改变进程的执行状态。

这里先开始介绍第一部分：实例2。

这是一个关于“信号的发送、接收以及处理”的实例。我们将以此来介绍对系统以及进程处理信号的过程。

有两个用户进程。一个进程用来接收及处理信号，名字叫做processsig。它所对应的程序源代码如下：

```
#include <stdio.h>

#include <signal.h>

void sig_usr (int signo) //处理信号的函数
{
    if (signo==SIGUSR1)
        printf ("received SIGUSR1\n") ;
    else
```

```

printf ("received%d\n", signo) ;

signal (SIGUSR1, sig_usr) ; //重新设置processsig进程的信号处理函数指针，以便下次使用

}

int main (int argc,char ** argv)

{

    signal (SIGUSR1, sig_usr) ; //挂接processsig进程的信号处理函数指针

    for (; )

        pause ();

    return 0;

}

```

另一个进程用来发送信号，名字叫做 **sendsig** 。它所对应的源代码如下：

```

#include <stdio.h>

int main (int argc,char ** argv)

```

```
{  
  
int pid,ret,signo;  
  
int i;  
  
if (argc != 3)  
  
{  
  
printf ("Usage: sensig < signo > < pid > \n") ;  
  
return -1;  
  
}  
  
signo=atoi (argv[1]) ;  
  
pid=atoi (argv[2]) ;  
  
ret=kill (pid,signo) ; //这里发送信号  
  
for (i=0; i<1000000; i++)  
  
if (ret != 0)  
  
printf ("send signal error\n") ;  
  
return 0;  
  
}
```

系统需要具备以下三个功能，以支持信号机制。

1.系统要支持进程对信号的发送和接收

系统在每个进程`task_struct`中都设置了用以接收信号的数据成员`signal`（信号位图），每个进程接收到的信号就“按位”存储在这个数据结构中。系统支持两种方式给进程发送信号：一种方式是一个进程通过调用特定的库函数给另一个进程发送信号；另一种方式是用户通过键盘输入信息产生键盘中断后，中断服务程序给进程发送信号。这两种方式的信号发送原理是相同的，都是通过设置信号位图（`signal`）上的信号位来实现的。

本实例将结合第一种方式，即一个进程给另一个进程发送信号来展现系统对信号的发送和接收。

2.系统要能够及时检测到进程接收到的信号

系统通过两种方式检测进程是否接收到信号：一种方式是在系统调用返回之前检测当前进程是否接收到信号；另一种方式是时钟中断产生后，其中断服务程序执行结束之前，检测当前进程是否接收到信号。

这两种信号检测方式大体类似。本实例将结合第一种方式来展现系统对进程接收到的信号的检测。

3.系统要支持进程对信号进行处理

系统要能够保证，当用户进程不需要处理信号时，信号处理函数完全不参与用户进程的执行；当用户进程需要处理信号时，进程的程序将暂时停止执行，转而去执行信号处理函数，待信号处理函数执行完毕后，进程程序将从“暂停的现场处”继续执行。

本实例将从“用户自定义的信号处理函数与进程进行绑定”、“系统对信号的预处理”和“信号处理完毕后进程现场的恢复”这三方面入手来展现系统是如何做到这些的。

下面我们就来介绍这两个进程都是如何开始运行的。现在用户处于shell界面下。

第一步：输入如下指令，运行processsig进程的程序。

```
[/usr/root]#./processsig&
```

```
<160> //这里可知processsig进程的进程号是160
```

```
[/usr/root]#
```

第二步：输入如下指令，运行sendsig进程的程序，发送信号SIGUSR1给processsig进程。

```
[/usr/root]#./sendsig 10 160//10代表SIGUSR1这个信号，160是processsig进程的进程号
```

```
received SIGUSR1
```

```
[/usr/root]#
```

现在，我们开始分别介绍这两个进程的执行情况。processsig进程是先开始执行的，我们先来

介绍它的执行情况。

8.2.1 信号的使用

1.processsig进程开始执行

processsig进程开始执行，要为接收信号做准备，具体表现为，指定对哪种信号进行什么样的处理。为此，进入main（）函数后，先要将用户自定义的信号处理函数与processsig进程绑定。用户程序是通过调用signal（）函数来实现这个绑定的。这个函数是库函数，它执行后会产生软中断int0x80，并映射到sys_signal（）这个系统调用函数去执行。sys_signal（）函数的功能是将用户自定义的信号处理函数sig_usr（）与processsig进程绑定。这意味着，只要processsig进程接收到

SIGUSR1信号，就调用sig_usr函数来处理该信号，绑定工作就是通过该函数来完成的。

进入sys_signal () 函数后，系统先要在绑定之前检测用户指定的信号是否符合规定。由于Linux 0.11中只能默认处理32种信号，而且默认忽略SIGKILL这个信号，所以只要用户给出的信号不符合这些要求，系统将不能处理。执行代码如下：

```
//代码路径: kernel/signal.c:

int sys_signal (int signum,long handler,long restorer)

{

    struct sigaction tmp;

    if (signum < 1||signum > 32||signum==SIGKILL) //经检测得知，信号符合规定

        return-1;
```

```
.....
```

```
}
```

检测完毕后，开始对processsig进程task_struct中的sigaction[32]进行设置。该管理结构有32个成员，正好对应默认的32种信号。sigaction[32]中每一个成员都会为每一种信号的处理提供一套服务。

执行代码如下：

```
//代码路径： kernel/signal.c:
```

```
int sys_signal (int signum,long handler,long restorer)
```

```
{
```

```
.....
```

```
if (signum < 1||signum > 32||signum==SIGKILL)
```

```
return-1;
```

tmp.sa_handler= (void (*) (int)) handler; //此时handler参数就是processsig进程中"signal (SIGUSR1, sig_usr)"这行代码中sig_usr函数的地址，这个绑定使得将来进程接收到信号，就由sig_usr这个函数来处理

```
tmp.sa_mask=0;
```

```
tmp.sa_flags=SA_ONESHOT|SA_NOMASK;
```

tmp.sa_restorer= (void (*) (void)) restorer; //这里绑定现场恢复函数

```
handler= (long) current->sigaction[signum-1].sa_handler;
```

current->sigaction[signum-1]=tmp; //sigaction[signum-1]这一项将为SIGUSR1信号提供服务

```
return handler;
```

```
}
```

此设置如图8-21所示。

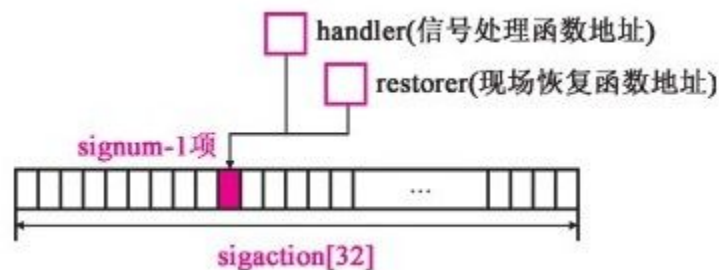


图 8-21 设置用户进程信号处理函数地址

值得注意的是，图8-21中的restorer（）函数也在sys_signal（）函数中绑定了。restorer（）函数的功能也很重要，后面我们会详细介绍。

processsig进程的状态及其代码在内存中的情况如图8-22所示，此时processsig进程处于就绪态。

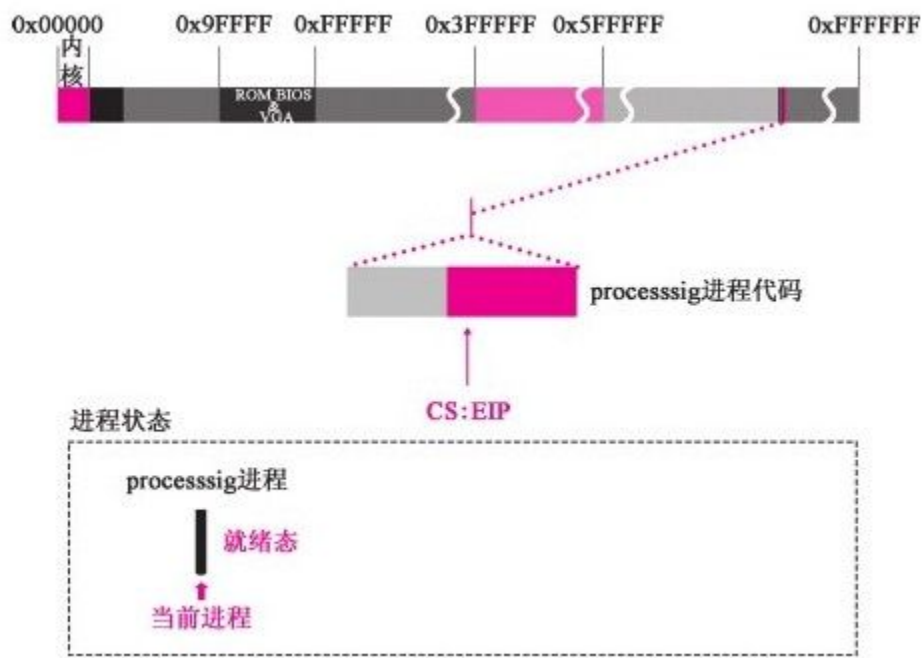


图 8-22 processsig进程及其代码在内存中的情况

2.processsig进程进入可中断等待状态

在processsig进程的程序中，为了体现信号对进程执行状态的影响，我们特意调用了`pause ()`函数。这个函数最终将导致该进程被设置为“可中断等待状态”。等到该进程接收到信号后，它的状态将由“可中断等待状态”转换为“就绪态”。

执行完`signal ()`函数后，将返回processsig进程的用户空间继续执行，调用`pause ()`函数。这个函数会映射到系统调用函数`sys_pause ()`中。执行代码如下：

```
//代码路径: kernel/sched.c:
```

```

int sys_pause (void)

{

    current->state=TASK_INTERRUPTIBLE; //将processsig进程设置
    为可中断等待状态

    schedule () ; //切换进程

    return 0;

}

```

processsig进程将被设置为可中断等待状态，
如图8-23所示。

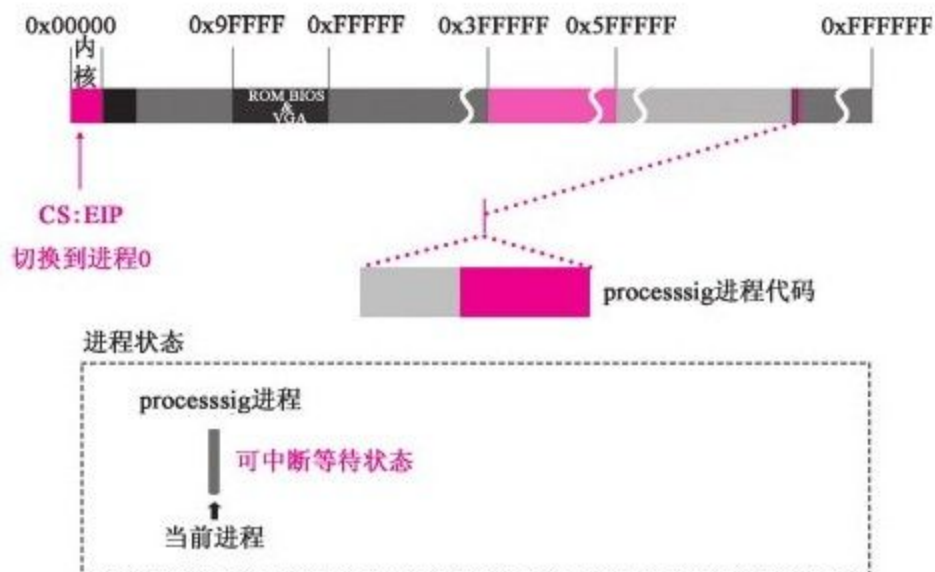


图 8-23 processsig进程进入可中断等待状态

3.sendsig进程开始执行并向processsig进程发信号

processsig进程暂时挂起，sendsig进程执行。sendsig进程就会给processsig进程发送信号，然后切换到processsig进程去执行。

sendsig进程会先执行“ret=kill (pid,signo)”这一行代码，其中kill是个库函数，最终会映射到sys_kill函数中去执行，并将参照“10”和“160”这两个参数给processsig进程发送SIGUSR1信号，执行代码如下：

```
//代码路径: kernel/exit.c:
```

```
int sys_kill (int pid,int sig)
```



```

{

.....

if (! pid) while (--p > &FIRST_TASK) {

if (*p && (*p) -> pgrp == current->pid)

if (err = send_sig (sig, *p, 1) )

retval = err;

} else if (pid > 0) while (--p > &FIRST_TASK) {

if (*p && (*p) -> pid == pid) //找到processsig进程

if (err = send_sig (sig, *p, 0) ) //此函数负责具体的发送工作

retval = err;

} else if (pid == -1) while (--p > &FIRST_TASK)

if (err = send_sig (sig, *p, 0) )

retval = err;

.....

}

```

//代码路径: kernel/exit.c:

```
static inline int send_sig (long sig, struct task_struct * p, int priv)
```

```

{

if ( ! p||sig < 1||sig > 32)

return-EINVAL;

if (priv|| (current->euid==p->euid) ||suser ( ) )

    p->signal|= (1<< (sig-1) ) ; //在processsig进程的信号位图
    (signal) 中找到SIGUSR1这一信号对应的位置，然后将其置1

else

return-EPERM;

return 0;

}

```

将SIGUSR1信号发送给processsig进程的过程及该过程对processsig进程管理结构中相应字段的影响如图8-24所示。

之后，就返回sendsig用户进程空间内继续执行随着时钟中断不断产生，sendsig进程的时间片

将被削减为0，导致进程切换，schedule（）函数开始执行。对应代码如下：

```
//代码路径： kernel/sched.c:

void schedule（void）

{

.....

for（p=&LAST_TASK； p> &FIRST_TASK； --p）

if（*p） {

if（（*p）->alarm&&（*p）->alarm<jiffies）{

（*p）->signal|=（1<<（SIGALRM-1））；

（*p）->alarm=0；

}

if（（（*p）->signal&~（_BLOCKABLE&（*p）->

blocked））&& //遍历到processsig进程后，检测到其接收的信号

（*p）->state==TASK_INTERRUPTIBLE） //processsig进程还是

可中断等待状态

（*p）->state=TASK_RUNNING； //将其设置为就绪态
```

}

.....

}

该过程如图8-25所示。

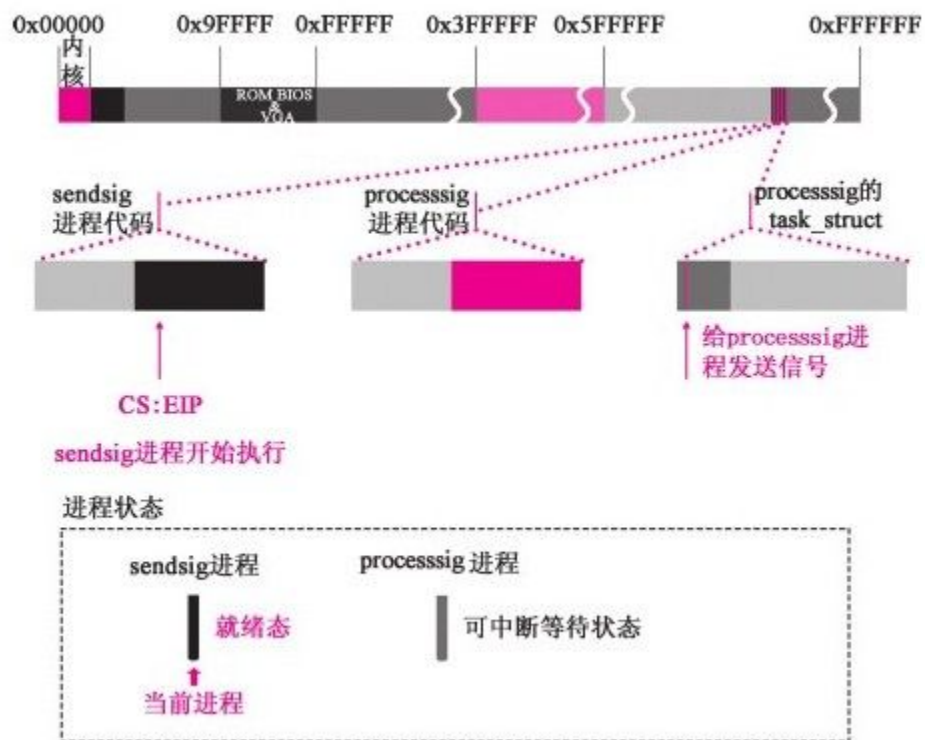


图 8-24 给processsig进程发送信号

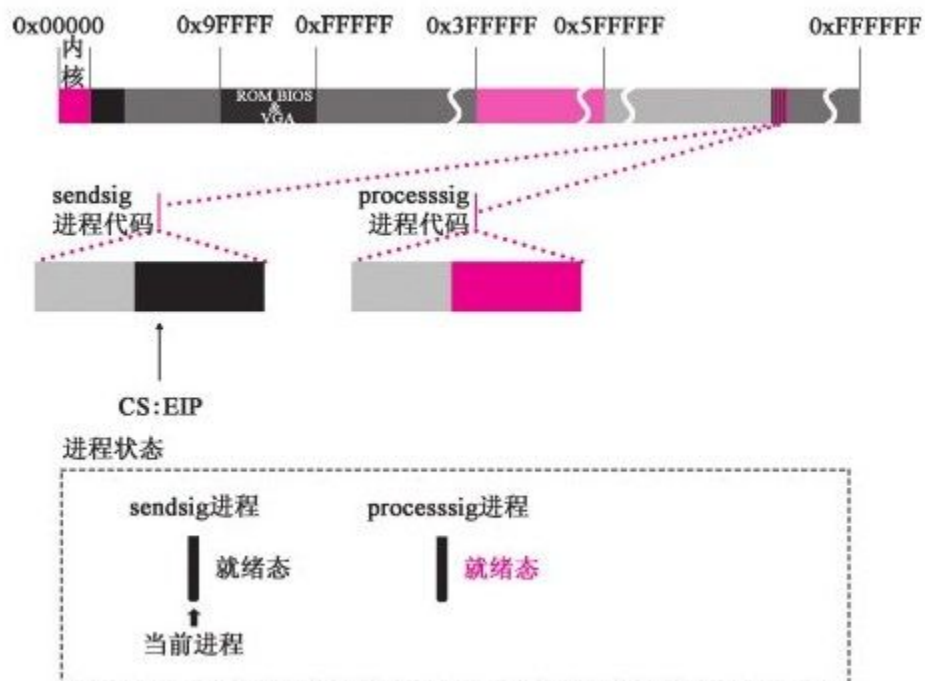


图 8-25 processsig进程收到信号后，被置为就绪态

等到第二次遍历时，就会切换到processsig进程去执行，执行代码如下：

//代码路径：kernel/sched.c:

```
void schedule (void)
```

```
{
```

.....

```
while (1) {
```

```
c=-1;
```

```
next=0;
```

```
i=NR_TASKS;
```

```
p=&task[NR_TASKS];
```

```
while (--i) {
```

```
if (! *--p)
```

```
continue;
```

```
if ( (*p) -> state==TASK_RUNNING && (*p) -> counter > c)
```

```
c= (*p) -> counter, next=i; //这时候processsig进程已经就绪了
```

```
}
```

```
if (c) break;
```

```
for (p=&LAST_TASK; p>&FIRST_TASK; --p)
```

```
if (*p)
```

```
    (*p) -> counter= ( (*p) -> counter > 1) +
```

```
    (*p) -> priority;
```

```
}  
  
switch _to (next) ; //切换到processsig进程去执行  
  
}
```

4.系统检测当前进程接收到信号并准备处理

processsig进程开始执行后，会继续在for循环中执行pause（）函数。由于这个函数最终会映射到sys_pause（）这个系统调用函数中去执行，所以当系统调用返回时，就一定会执行到

ret_from_sys_call: 标号处，并最终调用do_signal（）函数，开始着手处理processsig进程的信号。

执行代码如下：

```
//代码路径: kernel/system_call.s:  
  
.....  
  
ret_from_sys_call:
```

movl _current, %eax#task[0]cannot have signals

cmpl _task, %eax

je 3f

cmpw \$0x0f,CS (%esp) #was old code segment supervisor?

jne 3f

cmpw \$0x17, OLDSS (%esp) #was stack segment=0x17?

jne 3f

movl signal (%eax) , %ebx

movl blocked (%eax) , %ecx

notl %ecx

andl %ebx, %ecx

bsfl %ecx, %ecx

je 3f

btrl %ecx, %ebx

movl %ebx,signal (%eax)

incl %ecx

pushl %ecx


```
call _do_signal//准备处理信号
```

```
.....
```

5.系统检测信号处理函数指针挂接是否正常

现在开始介绍信号处理之前的准备工作。

进入do_signal（）函数后，先要对processsig进程的信号处理函数进行判定。我们在本节前面介绍过，processsig进程的信号处理函数指针被加载到了进程task_struct中的sigaction[32]结构中，如图8-26所示。

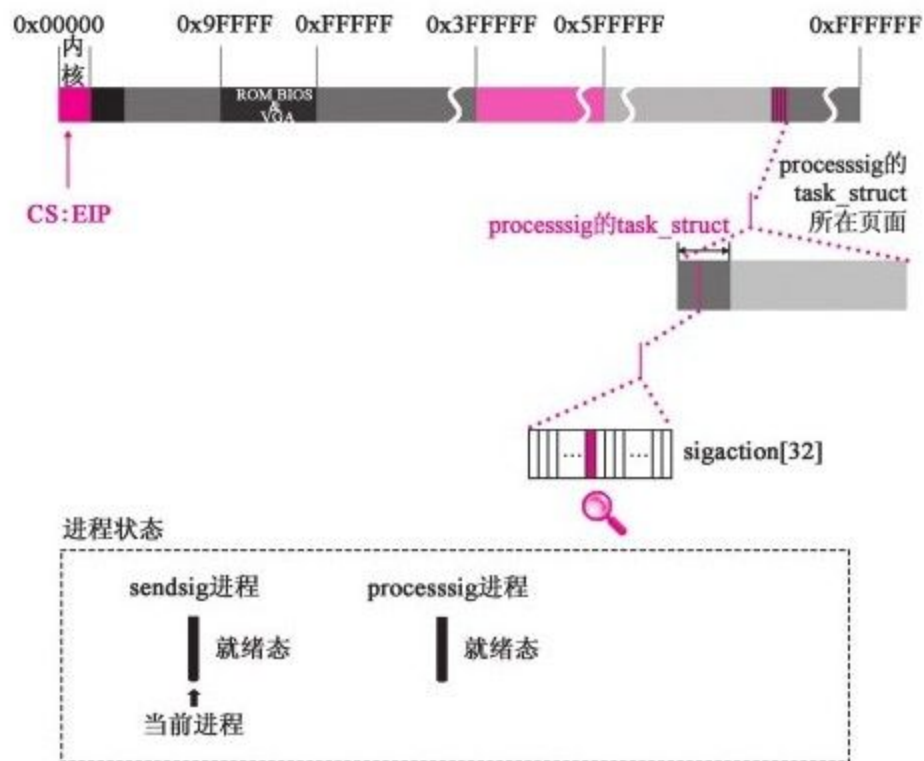


图 8-26 信号处理函数指针在sigaction结构中的位置

现在这个指针开始发挥作用，如果它为空，该进程就很有可能退出。当然，此时的情况是，这个指针肯定不为空，它指向了processsig进程的信号处理函数sig_usr（）。检测工作对应的代码如下：

//代码路径: kernel/signal.c:

```
void do_signal (long signr,long eax,long ebx,long ecx,long edx,  
long fs,long es,long ds,  
long eip,long cs,long eflags,  
unsigned long * esp,long ss)  
{  
.....  
struct sigaction * sa=current->sigaction+signr-1;  
.....  
sa_handler= (unsigned long) sa->sa_handler;  
if (sa_handler==1)  
return;  
if (! sa_handler) {//如果函数指针为空  
if (signr==SIGCHLD) //如果是SIGCHLD信号, 直接返回  
return;  
else
```

```
do _exit (1 << (signr-1)) ; //否则当前进程退出  
}  
  
.....  
}
```

6.调整processsig进程的内核栈结构，使其系统调用返回后，先执行信号处理函数

这里所做的准备工作的核心目的是对用户栈中的数据进行调整，使得此次系统调用返回后会“首先”执行processsig进程的“信号处理函数”，然后从用户进程“中断位置”继续执行。我们在本节前面所介绍的pause（）函数执行后产生int 0x80软中断的下一条指令处就是这个用户进程的“中断位置”（当然，如果不需要处理信号，直

接返回“中断位置”处就可以了，但现在要先处理信号问题，再回“中断位置”）。

软中断产生后，CPU将自动在当前进程“内核栈”中保存用户进程执行的“指令和数据”，其中包括EIP、CS、EFLAGS、ESP、SS这些寄存器的值。这样，只要系统调用返回，“内核栈”中这些数值反填回对应的寄存器，以此保证返回用户空间的“中断位置”处继续执行。

面对这种情况，Linux 0.11就在此次系统调用返回前，先把这些“内核栈”中保存的寄存器值备份在当前进程的“用户栈”中（内核有能力访问到所有物理内存，做这件事不成问题），然后对“内核栈”中的这些原有的寄存器值进行更改。这就使得系统调用函数返回后，首先根据“内核栈”中最

新更改的数据跳转到用户空间的信号处理函数处执行。进入用户空间后，“用户栈”就该发挥作用了，等到信号处理完毕后，再通过前面备份在用户空间的“指令和数据”，返回“中断位置”处执行。

这就是信号处理的全部策略。下面我们来看具体的实现代码：

```
//代码路径： kernel/signal.c:

void do_signal (long signr,long eax,long ebx,long ecx,long edx,
long fs,long es,long ds,
long eip,long cs,long eflags,
unsigned long * esp,long ss)
{
.....

if (sa->sa_flags&SA_ONESHOT)
```

```
sa->sa_handler=NULL;
```

```
* (&eip) =sa_handler; //调整内核栈中EIP位置, 使其指向  
processsig进程的信号处理函数sig_usr
```

```
longs= (sa->sa_flags&SA_NOMASK) ?7: 8;
```

```
* (&esp) -=longs; //对"用户栈"空间的栈顶指针ESP进行调整,  
使栈顶指针向栈底的反方向移动, 以便接下来在用户栈空间中备份数  
据
```

```
verify_area (esp,longs*4) ;
```

```
tmp_esp=esp; //以下是向用户栈空间中写入用于恢复现场的数据
```

```
put_fs_long ( (long) sa->sa_restorer,tmp_esp++) ;
```

```
put_fs_long (signr,tmp_esp++) ;
```

```
if ( ! (sa->sa_flags&SA_NOMASK) )
```

```
put_fs_long (current->blocked,tmp_esp++) ;
```

```
put_fs_long (eax,tmp_esp++) ;
```

```
put_fs_long (ecx,tmp_esp++) ;
```

```
put_fs_long (edx,tmp_esp++) ;
```

```
put_fs_long (eflags,tmp_esp++) ;
```

```
put_fs_long (old_eip,tmp_esp++) ;
```

```
current->blocked|=sa->sa_mask;
```

}

上述修改内核栈和用户栈的过程，以及用户栈空间和内核栈空间的数据调整前后的变化如图8-27和图8-28所示。

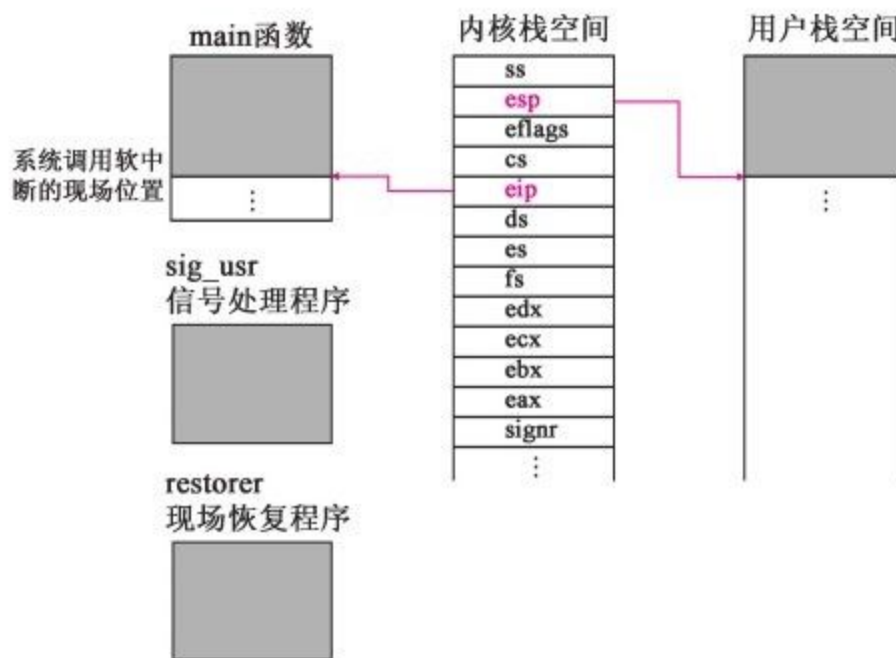


图 8-27 调整前的内核栈和用户栈的现场数据及其意义

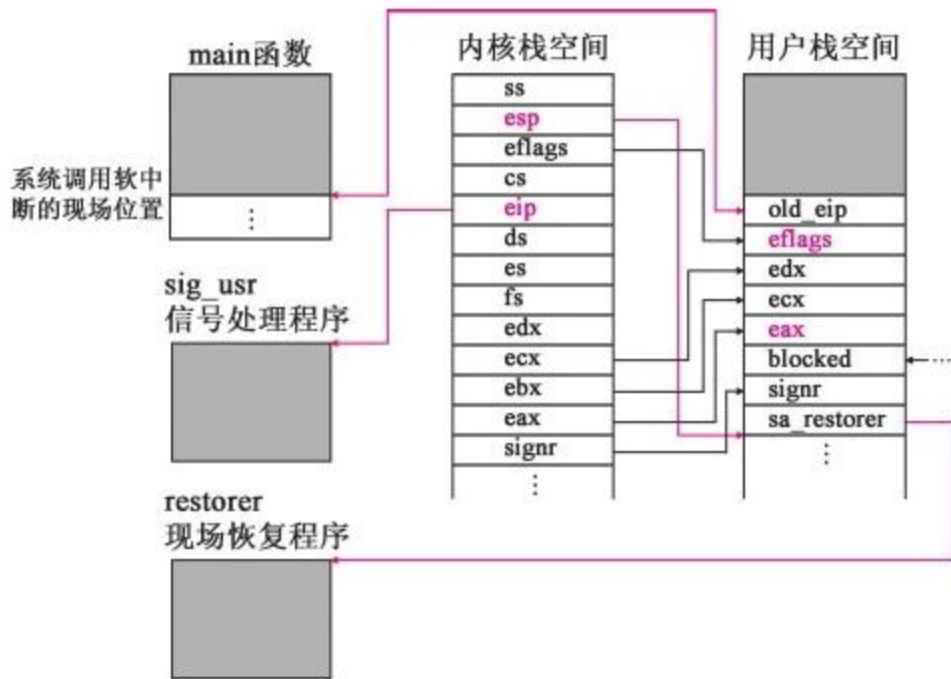


图 8-28 调整后的内核栈和用户栈的现场数据及其意义

信号预处理工作到这里已经完成。下面我们看这些数据在此次系统调用返回后都是如何被应用的，以及它们对“信号处理函数的执行”和“执行后进程现场的恢复”都产生了哪些影响。

前面已经将信号处理函数sig_usr与processsig进程进行了绑定，因此系统调用返回后，就会到processsig进程的sig_usr函数处执行，处理信号，函数执行结束后，会执行“ret”指令。ret的本质就是用当时保存在栈中的EIP的值来恢复EIP寄存器，跳转到EIP指向的地址位置去执行。于是此时处于栈顶的sa->sa_restorer所代表的函数地址值就发挥作用了，此时就应该跳转到sa->sa_restorer所代表的函数地址值位置去执行了。

本节前面讲到，还将一个叫做restorer的函数地址绑定在了sigaction[32]结构中。restorer是一个库函数的地址，它是由signal这个库函数传递下来的实参。这个库函数将来会在信号处理工作结束

后恢复用户进程执行的“指令和数据”，并最终跳转到用户程序的“中断位置”处执行。

现在信号已经处理完了，`restorer`函数开始工作。我们先来看一下这个函数的代码：

```
.globl ____sig_restore

.globl ____mask_sig_restore

____sig_restore:

    addl $4, %esp

    popl %eax//前面do_signal函数中最后设置的用户栈的内容，这里
    正好用来popl，用以恢复寄存器的值

    popl %ecx

    popl %edx

    popfl

    ret

____mask_sig_restore:
```

```
addl $4, %esp
```

```
call ____ssetmask
```

```
addl $4, %esp
```

popl %eax//前面do_signal函数中最后设置的用户栈的内容，这里正好用来popl，用以恢复寄存器的值

```
popl %ecx
```

```
popl %edx
```

```
popfl
```

```
ret
```

前面所介绍的do_signal（）函数调整栈空间的数值，正好在这里发挥作用。我们回顾一下代码，如下：

```
//代码路径： kernel/signal.c:
```

```
void do_signal (long signr,long eax,long ebx,long ecx,long edx,  
long fs,long es,long ds,
```

```

long eip,long cs,long eflags,

unsigned long * esp,long ss)

{

.....

put_fs_long ( (long) sa->sa_restorer,tmp_esp++) ;

put_fs_long (signr,tmp_esp++) ;

if ( ! (sa->sa_flags&SA_NOMASK) )

put_fs_long (current->blocked,tmp_esp++) ;

put_fs_long (eax,tmp_esp++) ;

put_fs_long (ecx,tmp_esp++) ;

put_fs_long (edx,tmp_esp++) ;

put_fs_long (eflags,tmp_esp++) ;

put_fs_long (old_eip,tmp_esp++) ;

current->blocked|=sa->sa_mask;

}

```

现场恢复后，就要返回现场执行。

注意看restorer函数最后一行汇编“ret”。由于ret的本质就是用当前栈顶的值设置EIP，并使程序跳转到EIP指向的位置去执行，很显然，经过一系列清栈操作后，当前栈顶的数值就是“put_fs_long (old_eip,tmp_esp++)”这行代码设置的。这个old_eip就是pause () 函数为了映射到sys_pause () 函数，产生软中断int0x80的下一行代码，即processsig进程的“中断位置”。所以，ret执行后，信号就处理完毕，并最终回到pause () 函数中去继续执行。

这就是Linux 0.11中信号处理的全部过程。

8.2.2 信号对进程执行状态的影响

下面将介绍本节的第二部分内容，即分别以进程的“可中断等待状态”和“不可中断等待状态”为例进行对比，体现信号对进程执行状态的不同影响。

可中断等待状态案例代码如下：

```
#include <stdio.h>

main ()

{

    exit () ;

}
```

shell进程创建了这个用户进程（该进程就自然成为shell进程的子进程）后，又被设置为可中断等待状态。现在，这个用户进程就要退出了。我们以此为例来介绍信号对进程执行状态的影响。

1.用户进程退出并向shell进程发送信号

用户进程先调用`exit ()`函数来处理自己退出前的一些事务，包括将自己的程序所占用的内存页面释放、解除该进程与所操作文件的关系等，之后，给shell进程发送“子进程退出”信号，通知shell进程，自己即将退出，最后将自己设置为僵死状态并调用`schedule ()`函数，准备进程切换，对应代码如下：

//代码路径: kernel/exit.c:

```
int do_exit (long code) //子进程退出
```

```
{
```

```
.....
```

```
if (current->leader)
```

```
kill _session () ;
```

```
current->state=TASK_ZOMBIE;
```

```
current->exit_code=code;
```

```
tell_father (current->father) ; //给父进程发信号
```

```
schedule () ; //进程切换
```

```
return (-1) ; /*just to suppress warnings*/
```

```
}
```

```
static void tell_father (int pid)
```

```
{
```

```
int i;
```

```
if (pid)
```

```
for (i=0; i<NR_TASKS; i++) { //寻找父进程, 即shell进程
```

```
if (! task[i])

continue;

if (task[i]->pid != pid)

continue;

task[i]->signal= (1 << (SIGCHLD-1)) ; //给shell进程发
送"子进程退出"信号

return;

}

.....

}
```

用户进程退出过程中发送信号以及将自身设置为僵死状态的过程如图8-29所示。



图 8-29 用户进程向shell进程发信号及将自身设置为僵死状态

2.shell进程被唤醒并调度执行

进入schedule () 函数后，先对所有进程进行第一次遍历，如果发现哪个进程接收到了指定的信号，而且该进程还是可中断等待状态，那么就将该进程设置为就绪态。系统通过遍历得知，

shell进程符合此条件，于是shell进程就被设置为就绪态，如图8-30所示。



图 8-30 shell进程设置为就绪态

对应代码如下：

```
//代码路径： kernel/sched.c:

void schedule (void)
{
.....

for (p=&LAST_TASK; p> &FIRST_TASK; --p)

if (*p) {
```

```

if ( (*p) -> alarm && (*p) -> alarm < jiffies) {

    (*p) -> signal |= (1 << (SIGALRM-1)) ;

    (*p) -> alarm = 0;

}

if ( ( (*p) -> signal & ~ (_BLOCKABLE & (*p) ->
blocked) ) && //检查进程是否接收到信号

    (*p) -> state == TASK_INTERRUPTIBLE) //检查进程是否为可
中断等待状态

    (*p) -> state = TASK_RUNNING; //如果条件同时符合，就将进
程设置为就绪态

}

.....

}

```

之后，第二次遍历所有进程，当前只有shell进程是就绪态，于是切换到shell进程去执行，执行代码如下：

//代码路径: kernel/sched.c:

```
void schedule (void)
```

```
{
```

```
.....
```

```
while (1) {
```

```
c=-1;
```

```
next=0;
```

```
i=NR_TASKS;
```

```
p=&task[NR_TASKS];
```

```
while (--i) {
```

```
if (! *--p)
```

```
continue;
```

```
if ( (*p) -> state==TASK_RUNNING&& (*p) -> counter>c)
```

```
c= (*p) -> counter,next=i;
```

```
}
```

```
if (c) break;
```

```
for (p=&LAST_TASK; p>&FIRST_TASK; --p)
```

```
if (*p)

    (*p) -> counter= ( (*p) -> counter > 1) +

    (*p) -> priority;

}

switch_to (next) ; //切换到shell进程去执行

}
```

3.shell进程执行，为子进程退出做最后 理

shell进程开始执行后，调用wait () 函数为子进程退出做处理，包括将子进程task_struct所占用的页面释放掉等，如图8-31所示。



图 8-31 shell进程处理用户进程退出后的善后工作

执行代码如下：

//代码路径：kernel/exit.c:

```
int sys_waitpid (pid_t pid,unsigned long * stat_addr,int options)
```

```
{
```

```
.....
```

```
repeat:
```

```
switch ( (*p) -> state) {
```

```
case TASK_STOPPED:
```



```

if ( ! (options&WUNTRACED) )

continue;

put_fs_long (0x7f,stat_addr) ;

return (*p) -> pid;

case TASK_ZOMBIE: //检测到子进程为僵死状态，将做如下处理

current->cutime+= (*p) -> utime;

current->cstime+= (*p) -> stime;

flag= (*p) -> pid;

code= (*p) -> exit_code;

release (*p) ;

put_fs_long (code,stat_addr) ;

return flag;

default:

flag=1;

continue;

}

if (flag) {

```

```
if (options & WNOHANG)

return 0;

current->state=TASK_INTERRUPTIBLE;

schedule ();

if ( ! (current->signal &= ~ (1 << (SIGCHLD-1)) ) ) //得
知接收到的信号为子进程退出信号

goto repeat;

else

return-EINTR;

}

return-ECHILD;

}
```

4.shell进程再次被挂起

之后shell进程继续执行，从tty0这个终端设备文件上读取数据。我们假设此时用户并没有通过

键盘输入任何信息，这样shell进程什么数据都没有读到，于是shell进程将被设置为可中断等待状态，等待着下次被唤醒，如图8-32所示。



图 8-32 shell进程再次进入可中断等待状态

由此可见，对处于可中断等待状态的进程而言，给它发信号，`schedule ()` 函数执行时会检测到它接收的信号和它的状态，并将其改设为就绪态，以此唤醒该进程。

下面介绍进程的不可中断等待状态。我们假设现在系统中有三个用户进程，分别是进程A、进程B和进程C，它们现在都处于就绪态，其中进程B是进程A的子进程，进程A正在运行。我们以此情景为例来介绍信号对进程执行状态的影响。

进程A和进程B案例程序如下：

```
main ()

{

char buffer[12000];

int pid,i;

int fd=open ("/mnt/user/hello.txt", O_RDWR, 0644) ) ;

read (fd,buffer,sizeof (buffer) ) ; //读文件

if ( ! (pid=fork ( ) ) ) {

exit ( ) ; //进程B（子进程）的代码

}
```

```
if (pid > 0)

while (pid != wait (&i) ) //等待子进程退出

close (fd) ;

return;

}
```

进程C案例程序如下:

```
main ()

{

int i,j;

for (i=0; i<1000000; i++)

for (i=0; i<1000000; i++)

}
```

1.进程A由于等待读盘而被挂起

进程A需要读硬盘，于是调用read（）函数产生软中断，并最终映射到sys_read（）函数去执行。经过一系列函数调用后，发送读盘命令，返回后，进程A被设置为不可中断等待状态。这是因为进程A下一步的执行需要此次读盘的数据来支持，在数据被读出之前，这个进程无论收到什么信号，都不能被唤醒。如果被唤醒了，它将操作缓冲区里面的数据，而此时缓冲区中的数据并没有从硬盘读出，这将引起数据混乱。这个过程如图8-33所示。

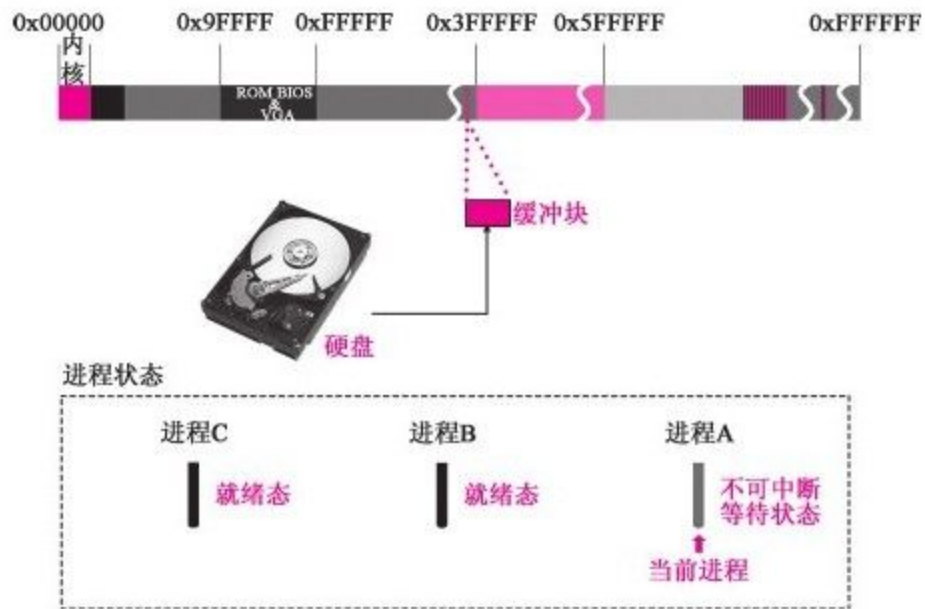


图 8-33 进程A挂起

进程A执行并最终挂起对应的代码如下：

```
//代码路径： fs/buffer.c:

struct buffer_head * bread (int dev,int block)

{

.....

if (bh->b_uptodate)

return bh;
```

```

ll_rw_block (READ,bh) ;

wait_on_buffer (bh) ; //检测是否需要等到缓冲块解锁

if (bh->b_uptodate)

return bh;

.....

}

static inline void wait_on_buffer (struct buffer_head * bh)

{

cli () ;

while (bh->b_lock) //缓冲块确实加锁了

sleep_on (&bh->b_wait) ; //只好将进程A挂起

sti () ;

}

//代码路径: kernel/sched.c:

void sleep_on (struct task_struct ** p)

{

.....

```



```
tmp=*p;

*p=current;

current->state=TASK_UNINTERRUPTIBLE; //将进程A设置为不可中断等待状态

schedule ();

if (tmp)

tmp->state=0;

}
```

2.进程A切换到进程B执行

之后也要调用`schedule ()` 函数，最终会切换到其他进程执行。我们假设切换到进程A的子进程，即进程B去执行。进程B执行后，准备退出，于是进程B被设置为僵死状态，之后给进程A发信号，通知进程A自己将要退出了，最后调用

schedule () 函数，准备进程切换，如图8-34所示。



图 8-34 进程B退出并给进程A发信号

执行代码如下：

//代码路径： kernel/exit.c:

int do_exit (long code) //进程B退出

{

```

.....

if (current->leader)

kill_session ();

current->state=TASK_ZOMBIE;

current->exit_code=code;

tell_father (current->father); //给父进程发信号

schedule (); //进程切换

return (-1); /*just to suppress warnings*/

}

static void tell_father (int pid)

{

int i;

if (pid)

for (i=0; i<NR_TASKS; i++) { //寻找父进程, 即进程A

if (! task[i])

continue;

if (task[i]->pid !=pid)

```

```
continue;

task[i]->signal= (1<<(SIGCHLD-1)) ; //给进程A发送"子进
程退出"信号

return;

}

.....

}
```

3.进程A虽收到信号，但无法唤醒

进入schedule（）函数后，第一次遍历所有进程。此时进程A虽然接收到了信号，但由于它是不可中断等待状态，所以并不会将它改设为就绪态，于是此次只能切换到进程C去执行，如图8-35所示。

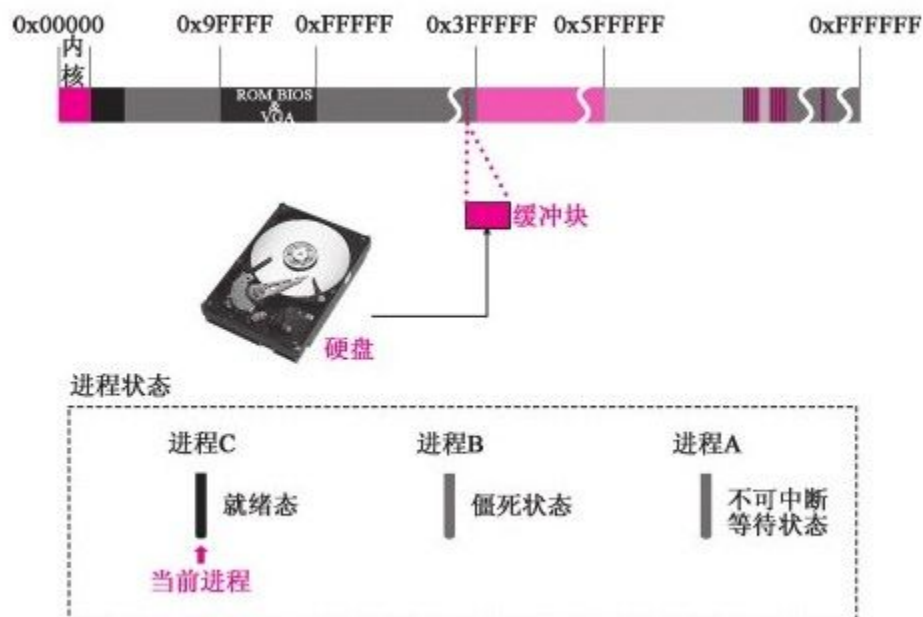


图 8-35 进程A为不可中断等待状态，无法用信号唤醒

执行代码如下：

//代码路径：kernel/sched.c:

```
void schedule (void)
```

```
{
```

```
.....
```

```
for (p=&LAST_TASK; p> &FIRST_TASK; --p)
```

```

if (*p) {

if ( (*p) -> alarm && (*p) -> alarm < jiffies) {

    (*p) -> signal |= (1 << (SIGALRM-1)) ;

    (*p) -> alarm=0;

}

if ( ( (*p) -> signal & ~ (_BLOCKABLE & (*p) ->
blocked) ) && //检查到进程A确实接收到信号

    (*p) -> state == TASK_INTERRUPTIBLE) //进程A状态为不可中
断等待状态

    (*p) -> state = TASK_RUNNING; //不会执行这里

}

.....

}

```

4. 由于外设数据读取完成，进程A被唤醒

进程C执行了一段时间后，进程A指定的数据已经从硬盘上读出，于是硬盘中断服务程序会将

进程A强行设置为就绪态（这也是将处于不可中断等待状态的进程改设为就绪态的唯一方法），如图8-36所示，执行代码如下：

```
//代码路径: kernel/blk_dev/blk.h:

extern inline void end_request (int uptodate)

{

    DEVICE _OFF (CURRENT->dev) ;

    if (CURRENT->bh) {

        CURRENT->bh->b_uptodate=uptodate;

        unlock_buffer (CURRENT->bh) ; //缓冲块解锁

    }

    if (! uptodate) {

        printk (DEVICE_NAME"I/O error\n\r") ;

        printk ("dev%04x,block%d\n\r", CURRENT->dev,

            CURRENT->bh->b_blocknr) ;
```

```

    }

    .....

}

extern inline void unlock_buffer (struct buffer_head * bh)

{

    if (! bh->b_lock)

        printk (DEVICE_NAME": free buffer being unlocked\n") ;

        bh->b_lock=0;

        wake_up (&bh->b_wait) ; //把等待缓冲块解锁的进程唤醒，即
唤醒进程A

}

```

这样进程A就具备执行能力，但这并不等于进程A马上就执行。硬盘中断服务程序返回后，仍然是进程C继续执行，如图8-36所示。

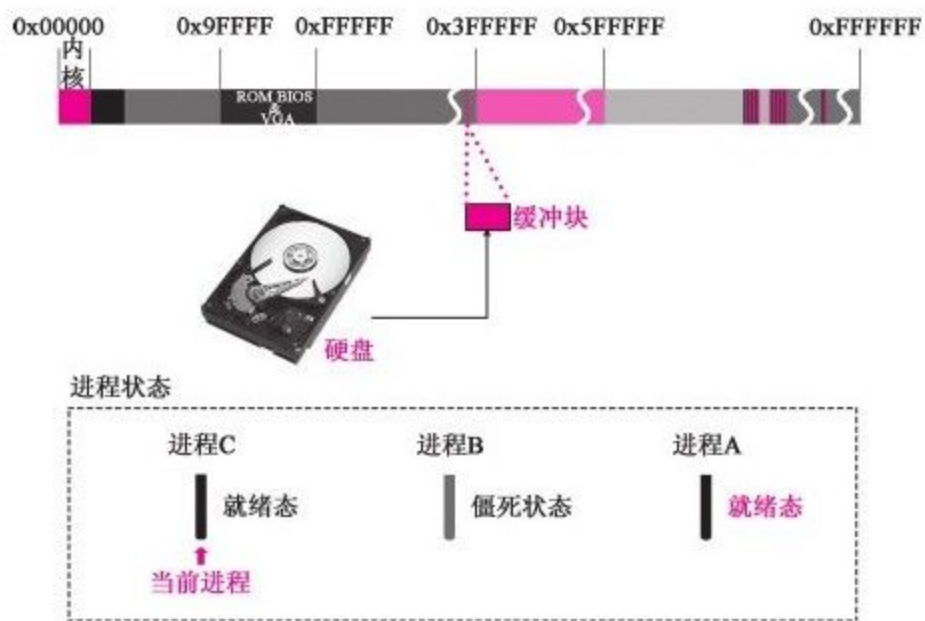


图 8-36 进程A被唤醒

5.切换到进程A执行并处理信号

进程C的时间片用完了，又要进行进程切换。这里进入`schedule()`函数后，发现只有进程A是就绪态，于是切换到进程A去执行。进程A开始执行后，先要对刚才从硬盘上读出来的数据进行处理，至此，`sys_read()`函数执行完毕。此时，软中断准备返回，在返回之前，先要检查一

下进程A是否有接收到任何信号。果然，检查到进程A接收到了信号，于是将处理该信号的服务程序入口地址进行处理，以便于一旦此次软中断返回，就由信号处理程序处理该信号（现在的情况是，要将进程B退出的善后事务彻底处理完），这个过程如图8-37所示。

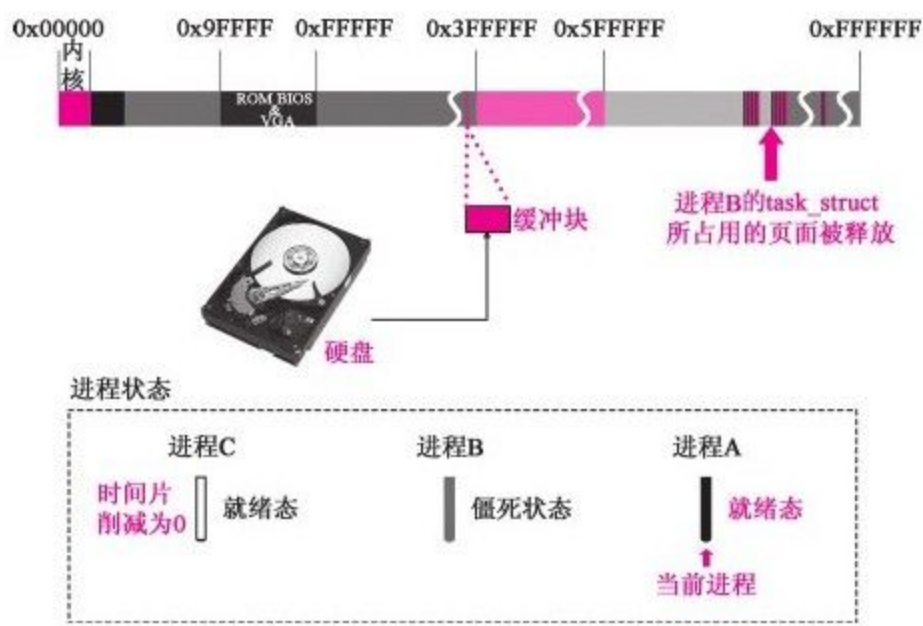


图 8-37 进程A执行并处理信号

由此可见，对处于不可中断等待状态的进程而言，除直接将其设置为就绪态之外，没有任何办法将它的状态改设为就绪态，是否接收信号都没意义。

8.3 本章小结

进程间通信本应是操作系统比较难以掌握的内容，但Linux 0.11的进程间通信设计的比较简单。俗话说，麻雀虽小，五脏俱全。读者可以依托这个虽然简单却可实际运行的进程间通信模型，进一步深入理解高版本的进程间通信问题。

信号是操作系统中很重要的概念。本章详细讲解了信号机制，并提出了信号是仿软中断的理解。

第9章 操作系统的设计指导思想

普天之下，莫非王土；率土之滨，莫非王臣。

——《诗经小雅》

前8章详细分析和讲解了Linux操作系统的运行原理与工作机制。本章将尝试从设计者的视角探讨操作系统的设计指导思想。

9.1 运行一个最简单的程序，看操作系统为程序运行做了哪些工作

透彻理解操作系统的好方法之一就是查看当一个最简单的程序在计算机上运行时，操作系统

都做了些什么。

我们以一个C语言版的“hello world”为例：

```
#include <stdio.h>

void main ()

{

printf ("hello world\n") ;

}
```

这段程序被编译、链接之后会生成一个可执行文件。我们在Linux操作系统上运行这个程序，最终会在屏幕上输出“hello world”。表面上看，在屏幕上显示的“hello world”都是我们写的程序的功劳，其实我们写的程序只起到了很小的作用。很明显的是，这个程序里使用了C语言的库函数

`printf`。因为本书的主题是操作系统，所以我们暂不讨论库函数。

我们粗略地浏览一下Linux 0.11为hello程序的运行都做了些什么。下面的小字是对操作系统所做的工作的概要描述，涉及操作系统的代码在1万行以上，除进程间通信之外，几乎涉及操作系统的方方面面。我们写这些文字的目的，是想让读者对运行一个最简单的程序，操作系统都做了哪些工作有一个直观的了解。

案例描述：硬盘上有一个名叫hello的可执行文件，该文件的源程序如前。

现在系统已经处于怠速状态了，用户准备通过键入一条指令`./hello`，使硬盘上该文件中的程序

加载并执行，最终将hello world这个字符串显示出来。

第一步：用户输入命令，shell进程被唤醒，对命令进行解析。

为实现这一步，系统将至少进行如下准备工作。

(1) 用户敲击键盘后，键入的信息记录在终端设备文件（tty0）上。

如果系统要以文件的形式对终端设备进行操作，首先要构建一整套文件系统，之后加载该文件系统，以便在此基础上对文件进行操作。这套文件系统包括“超级块”、“逻辑块位图”、“i节点位图”、“文件i节点”、“数据块”等。其次，还要根据

文件的不同功能对数据进行分类，包括普通文件、设备文件、目录文件等。tty0就属于设备文件。有了这些准备才有可能对终端设备文件tty0进行操作。

(2) 敲击键盘后，还要产生键盘中断信号，系统要能够对键盘中断信号进行处理。

首先，这个中断信号会通过可编程中断控制器8259A，所以要对8259A这个中断控制器进行设置；然后，信号会被传达给CPU,CPU要通过中断描述符表寄存器（IDTR）找到内存中的中断描述符表，再通过搜索中断描述符表找到键盘中断处理程序，并执行该程序。要实现这些操作，就要构建一整套中断服务体系，其中包括对中断描述符表寄存器（IDTR）进行设置和建立一个中断描

述符表，用以和中断服务程序相挂接，然后还要编写中断服务程序，以便能够为具体的中断服务。此外，还要将这些中断服务程序与中断描述符表相挂接。

(3) 中断服务程序开始执行后，唤醒shell进程，之后，通过进程调度机制，由进程0切换到shell进程去执行。

这需要系统建立一整套进程管理机制。就shell来说，要创建进程并加载shell程序，这样才能构建人机交互界面；同时，还要创建一个进程0，并在其他进程都不处于就绪态时切换到进程0去执行，而且一旦有进程被唤醒，就又立即切换到该进程执行。这个机制要适用于操作系统中的所有进程。既然要支持多进程执行，就还要设计

一套进程轮询机制，即产生时钟中断，导致进程切换。这个机制里面又有很多的问题需要考虑，比如时钟中断服务程序的设计和8253定时器的设置，等等。

（4）**shell**进程通过执行自己的程序从**tty0**这个终端设备文件上读取用户键入的指令信息，然后解析该指令，并准备进行相应的处理。当然，这条指令不是敲击一次键盘就能输入的。每次敲击键盘，都会重复上述动作，然后**shell**进程再次睡眠，并等待下一次键盘中断的产生。

到这里为止，系统仅仅是对用户键入的命令进行响应，正式的处理还没有开始。以上介绍的这些准备工作，都只是针对具体的步骤进行了最简单的介绍。这些准备工作的背后，又有着更多

的准备工作。严格来讲，本书第1、2、3、4章中所介绍的准备工作，几乎都要派上用场，才能执行这第一步。

第二步：**shell**程序解析出用户命令后，调用**fork**函数创建一个用户进程，以便对**hello world**文件的程序进行控制。

系统在这里至少要为用户进程创建一套进程管理结构。每套进程都要有一套这样的结构，以便控制将来加载的程序。这套结构十分复杂，包括时间片、优先级、进程状态、进程对应的文件、进程的任务状态描述符表（**TSS**），以及进程的局部数据描述符表（**LDT**），等等。其中的每一项又与系统的运行有着千丝万缕的关系。比如说**TSS**，它里面存放着当前进程运行时所有寄

寄存器中的数据，一旦发生进程切换，系统就将当前各个寄存器中的数据存储在TSS中，同时用即将切换到的进程中的TSS中的数据来设置各个寄存器中的值，最后再切换。可见，这个TSS中的数据是进程切换的根本保障。再比如LDT，它里面存放着当前进程的代码段描述符和数据段描述符，这两个描述符都直接控制着进程所控制的程序，而进程运行的根本目的就是执行用户的程序。

另外，每个进程都会有TSS和LDT，为了便于管理，还需要再设计一套数据结构，就是全局描述符表（GDT）。所有进程的TSS和LDT的索引都存放在这个GDT中。系统为了方便操作GDT，并进一步操作LDT和TSS，还要对CPU中

关于这三个表的专用寄存器进行设置，它们就是全局描述符表寄存器、局部数据寄存器和任务状态寄存器。

但仅有这些，还是远远不够的。系统启动之初是实模式，各个段寄存器中都是实际的地址值，直到进入保护模式，段寄存器中的数值才变成了段选择符，这样GDT才能参与应用，所以系统还要为实模式到保护模式的转换做全方位的准备工作。

以上这些只是针对TSS和LDT进行的展开分析。进程管理结构中其他成员与系统之间同样有着紧密的关系。例如，进程调度的最基本方式就是通过时间片轮转，而时间片轮转的最重要的参考数据就是当前进程的时间片。再例如，只有进

程才能够操作文件，所以进程就要与文件全面建立关系，包括文件的i节点、文件管理表中的表项、进程自身的文件管理指针表，等等。

创建进程，就必须创建进程管理结构，而进程管理结构中的成员一个都不能少，都要创建并设置。除了进程管理结构，创建进程的时候，还要为新进程复制页表和创建页目录项。这些都与内存页面的应用有直接关系，而内存的应用策略又是整个操作系统中最复杂的应用策略之一。

第三步：新进程创建完毕后，加载hello world文件对应的程序。

要完成这一步，进程就要在两方面进行全方位的准备：一方面是文件，另一方面是内存。

hello world程序一定是以可执行文件的方式存储在硬盘上的，所以，在文件加载之前一定要检测文件是否可用，主要表现在对文件i节点的检测和对文件头的检测两方面。i节点是文件的管理信息。只要涉及i节点，就一定离不开对i节点的查找，于是就要解析文件路径、操作目录文件和目录项、操作i节点表等，一件事情都不能少做。文件头存储在数据块中，要操作数据块又离不开逻辑块位图的支持，这样一来，整个文件系统中涉及的全部内容都要用到了。

具备了载入文件的条件后，就要将hello world文件载入内存中了。这样系统就要解决所有与内存相关的问题，包括要与原来进程共享的页面解除关系，这就涉及页面引用计数、页面三级

管理机制（页目录表、页表、页面）、页面数据（只读/可读可写）等一系列问题，系统就要为此建立页写保护等机制来解决这些问题。

但仅仅解决这些问题还远远不够，程序的加载也是很讲究策略的，其中最重要的就是缺页中断机制，即必须根据需求来分析是不是需要申请新的页面来加载程序的内容。为此又要对许多数据进行判断，这样才能确定加载的必要性，比如线性地址所对应的物理地址是否被映射到了线性地址空间内等，这就需要一套物理地址到线性地址的映射方案。另外，缺页中断机制的设计也很有讲究，缺页并不等于一定要把外设的程序加载进来。比如，由压栈导致的缺页同样要申请新的页面来载入数据，但这与外设一点关系都没有。

这些都是缺页中断机制设计时需要全面考虑的问题。

总之，`hello world`程序的加载，几乎涉及文件管理与内存管理的各方面。而且，以上所述还仅仅是针对`hello world`这一个进程加载所进行的最基本的介绍。`Linux`是要支持多进程执行的，每个进程都有可能加载自己的程序，而文件和内存又是所有进程可以共用的资源，它们之间还存在着更为复杂的管理关系。比如，两个进程加载同一个`hello world`文件时，涉及要不要共享，如何共享，共享后页面的引用计数如何计算，读写属性如何确定，等等。

第四步：`hello world`程序开始执行，将“`hello world`”字符串显示在屏幕上。

hello world程序加载进内存后就要开始执行了。这个程序比较简单，就是将hello world这个字符串显示到屏幕上。但是，即便如此，系统也要为此做很多的工作。其中最主要的就是关于显示方面的工作，比如，显卡属性如何确定，显卡是单色还是彩色；显存位置如何确定，显示在屏幕上的位置又如何确定；如果字符数量过多，要不要滚动显示，如何滚动显示，等等。这些问题都要操作系统来做，而且直接与显示器的底层交互。

从以上这些小字中，我们不难看出，即使是运行一个最简单的程序，操作系统也要做非常多的工作。我们反向思考一下，似乎也可以得出下面的结论：如果没有操作系统，即使是在屏幕上

显示一行“hello world”，我们都要写大量复杂的、具有操作系统所具备的功能的程序。毫不夸张地说，没有操作系统，我们甚至无法把程序加载到计算机中，更谈不上得到运行结果了。

那么，操作系统究竟为应用程序的运行都做了些什么？

经过综合分析，我们归纳出，操作系统的一部分任务是为应用程序的运行提供使用硬盘、显示器、键盘等外设的基础程序，或者说操作系统为应用程序的运行提供了对外设的支持。如果操作系统不写这些支持程序，应用程序就必须写这些程序，而且所有应用程序都要写的这部分程序的内容也都差不多。所以，我们也可以把操作系统看成所有应用程序共有的部分。

像Linux这样的现代操作系统，不仅为应用程序提供了对外设的支持，还支持多个程序同时运行。这就要求操作系统不但要支持外设，还必须对运行的多个程序进行有效的组织、管理和协调，防止某个程序独占CPU、内存、外设等资源，使得其他程序无法正常运行。此外，还要防止正在运行的程序之间相互读写和相互覆盖，确保所有程序正确运行。最关键的是，操作系统不能被应用程序直接读写，更不能被应用程序覆盖。

9.2 操作系统的设计指导思想——主奴机制

以上这些看似合情合理的要求背后似乎隐藏着这样一个问题：应用程序是程序，操作系统也是程序，操作系统程序凭什么能对应用程序进行组织、管理和协调而不受应用程序损害呢？

我们认为凭的是特权机制。要想让操作系统做到能够对应用程序进行组织、管理和协调，同时又不受到损害，最有效的方法就是使操作系统与应用程序之间、应用程序与应用程序之间进行有效的分离，同时要做到操作系统能随意访问应用程序，而应用程序不能访问操作系统，应用程序之间也不能互相访问。

这意味着，操作系统必须能够做到，如果它要让某个应用程序在内存中的什么位置运行，该应用程序就得老老实实地在那里运行；操作系统应在内存中为应用程序划出清晰的边界，应用程序不能越雷池半步。操作系统允许应用程序占用CPU运行多长时间，应用程序只能运行多长时间，运行完这点时间后，还得将CPU的使用权规规矩矩交还给操作系统，没有权利私自扣留CPU的使用权。如果某个应用程序要想使用外设，不能直接向外设伸手，得向操作系统请示和申请。如果操作系统认为可以让这个应用程序使用外设，就让它使用；如果操作系统认为不应该让这个应用程序使用外设，就不让它使用.....

在这样的特权机制下，操作系统和应用程序的关系就成了主子和奴才的关系。为了便于记忆，我们把这种特权机制称为主奴机制。

9.2.1 主奴机制中的进程及进程创建机制

1. 程序边界与进程

为了实现主奴机制，首先要在操作系统内核程序与应用程序之间、应用程序与应用程序之间建立有效的边界。

现实生活中的物体，大部分种类都是有自然边界的。比如我们周围的房子、桌椅板凳都有天然的边界；我们自身有皮肤，这是我们的天然边界。这些天然的边界可以有效地防止人与人之间、物与物之间、人与物之间的融合，以保持独

立、完整的形体，同时保持了独立的、完整的特性。现实生活中也有一些物质没有天然的、确定的边界，比如气体和液体等。不同的气体之间因为没有边界，很容易相互融合，不分你我。我们想要盛水，最有效的方法就是用杯子或瓶子之类的有边界的容器。

程序代码在计算机中的情况与气体和液体的情况类似，也没有天然的、确定的边界，这就需要操作系统人为确定边界，起到类似于容器的分离和承载的作用。为此，现代操作系统的设计者提出了进程的概念，用`task_struct`数据结构来实现明确地划分边界的作用。`task_struct`是进程最主要的标志。从操作系统的角度看，进程就是运行中的接受操作系统组织、管理和协调的程序。

2.进程创建

从技术上讲，创建进程的方法不止一种，Linux操作系统的进程创建采用的是对象创建模式。对象创建就是用已有的对象创建新的对象，用已有的进程创建新的进程，也就是所谓的父子进程创建机制。从本质上讲，创建进程最主要的就是创建task_struct。父子进程创建的主要机制就是从父进程的task_struct复制一份作为子进程的task_struct。

从逻辑上很容易反推出，父子进程创建机制意味着最初的父进程必须独立存在，这就是进程0。不难理解，进程0不能由父子进程创建机制创建，所以只能由操作系统设计者手工编写进程0的

`task_struct`。有了进程0，父子进程创建机制就可以用进程0作为父进程创建子进程。

有了进程，就有了主奴机制的组织、管理和协调的对象。下面我们将分析操作系统中的主奴机制是如何实现的。

9.2.2 操作系统的设计如何体现主奴机制

操作系统内核与用户进程之间的关系应该设计为主奴关系，以实现主奴机制。实现了主奴机制，操作系统才能稳定地运行。

操作系统的设计者使用了一整套设计方案来实现主奴机制。下面我们从操作系统设计者的角度，分三方面来剖析操作系统的设计思想中是如何体现主奴机制的。第一方面，进程调度体现的主奴机制；第二方面，内存管理体现的主奴机制；第三方面，文件系统体现的主奴机制。

1.操作系统在进程调度中体现的主奴机制

操作系统进行进程调度时，对待内核和进程的方式是截然不同的，准确地说，进程调度就是内核操作的。当发生时钟中断时，触发调度程序，调度程序判断当前进程的时间片是否用完，如果用完，马上进行调度，不论这个进程的工作是否完成，立即挂起这个进程，调度其他进程运行。如果是内核，调度程序判断确认后就返回，内核继续运行，直到工作完成为止，不论占用CPU的时间有多长，都是如此，所有的用户进程都要停下来，一直等待内核运行结束。由此可见，系统掌握了“进程调度的权利”后，该权利只针对它的奴才——进程，而不能针对主子本身——内核。

Linux操作系统每次分配给进程的运行时间是由若干时间片组成的，分配多少时间要由内核决定，内核决定多少就是多少，进程想要增加时间片，连商量的机会都没有。一旦时间片用完，CPU的使用权就要交还内核。值得注意的是，这个“交还”不是通过协商或轮流“坐庄”，而是强行收回！如果进程运行结束，就算还有剩余的时间片，操作系统也会收回CPU的使用权，不会等待。

如果不采用主奴机制，而是把进程调度设计成进程自觉、主动地将执行权限上交至操作系统，那么，进程什么时候上交CPU的使用权以及是否上交，直接取决于进程的程序设计，操作系统几乎无法控制。更可怕的是，如果进程是恶意

程序，或者发生异常僵死，操作系统很可能永远无法回收CPU的使用权，会导致整个系统瘫痪。

主奴机制的设计保证了应用程序占用CPU资源的情况最多只影响操作系统分配给它的那些时间片。这些时间片用完后，执行权限自然回归操作系统。回归之后，操作系统就可以处理进程，甚至将出错的进程强行退出。

具体的技术细节在第6章已经详细讲解过了。

2.操作系统在内存管理中体现的主奴机制

前面讲过，进程最主要的标志就是task_struct数据结构。在task_struct数据结构中，明确地定义了进程的边界，任何未经允许的跨越边界行为都将被制止。有了清晰的边界，就保证了进程不能

直接越界访问操作系统内核，也保证了进程间不能直接相互访问。这是主奴机制的体现。

Linux 0.11的内核和用户进程都采用了分页机制，但是采用的管理数据却是两套：一套是针对内核使用的，分页范围是0~16 MB的全部内存空间；另一套是针对用户进程的，分页范围仅限于1~16 MB空间（1 MB以内的是内核空间），如图9-1所示。

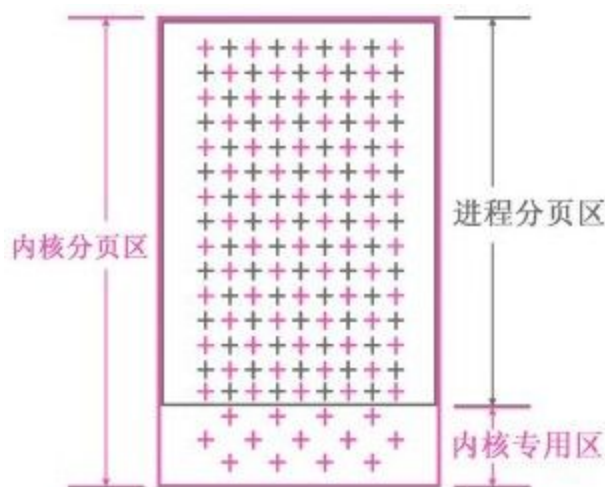


图 9-1 内核与用户区分页示意图

从图9-1中可以清楚地看出，只要操作系统内核代码在内核专用区，进程是无论如何也访问不到的，而内核的访问范围却囊括整个内存空间。我独占的地方你来不了，你的地方我随便去，因为你的地方就是我的地方。真是“普天之下，莫非王土”，其结果必然是“率土之滨，莫非王臣”！

此外，用户进程只能面对一个逻辑地址，不能直接使用物理地址，需要使用内存的时候，转化为一个线性地址，再根据第二套1~16 MB的管理数据方案转化为实际物理地址。内核将整个内存划分为统一大小的内存块——页。进程运行时，操作系统给进程分配页。如果操作系统给进程分配了两个以上的页，这些页不一定是相邻的，而且通常是不相邻的。这些页具体分配在内

存的什么地方，进程是不知道的；分了几页，进程也不知道。准确地说，进程甚至都感觉不到分页。在进程看来，它使用的是一个连续的逻辑地址内存空间。

进程不知道自己在哪儿，更不可能知道别的进程在哪儿，根本不可能进行进程间的互访。内核代码肯定放在内核专用区里，从图9-1可以明显地看出位置在进程的访问极限之外，所以进程不可能访问内核。进程存储区在内核的访问范围之内，内核就可以随意访问进程的内存空间。这是典型的主奴关系！

具体的技术细节在第3章和第6章已经详细讲解过了。

3.操作系统在文件系统中体现的主奴机制

以写文件时申请磁盘空间为例。当用户需要向磁盘写文件时，首先需要向内核提出申请，说明自己是哪个进程、自己需要的资源大小，以及资源的读写权限；内核接到申请之后，会结合当前的磁盘已占用资源、缓冲区的实际情况决定是否立即满足该用户进程的请求。如果有多个进程都在申请资源，内核会决定让某一个进程先获得资源，而让其他进程处于等待状态，并且要对等待队列的排序做出管理，以达到内核认为的最好标准。如果当前的资源已经无法满足要求，内核还会驳回进程的申请。这里也充分体现了内核和用户进程的主奴机制。用户进程的工作一旦涉及底层，是没有权利直接向资源伸手的，需要先向

内核“提申请”、“打报告”。内核掌控着各种硬件资源，负责对申请资源的众多进程进行组织、管理和协调。

具体的技术细节在第5章中有详细的讲解。

9.3 实现主奴机制的三种关键技术

上面我们从三方面详细分析了主奴机制的设计指导思想在操作系统中的体现。下面，我们将讲解操作系统的设计者是靠什么实现主奴机制的。我们认为，主要依靠三项关键技术：保护和分页、特权级、中断。这三项技术有一个共同特点，就是依托CPU提供的硬件机制。

9.3.1 保护和分页

我们在第1章就讲过，Linux 0.11打开了PE和PG，即打开了保护模式和分页机制。打开保护模式后，CPU的寻址模式发生了实质性的变化。以代码寻址为例，实模式时是CS: IP，在保护模式

下IP变为EIP，更关键的变化是CS由直接的代码段基址变为代码段选择符，通过解析代码段选择符可获得GDT中指定的代码段描述符，进一步解析才能获得代码段的基址。

还有两个变化是深刻的：一个是段限长，另一个是特权级。

实模式下的CS虽然是代码段的段基址，但CS只是负责看管代码段的起始位置，Intel的CPU没有设计负责看管代码段结尾地址的段尾寄存器。虽然有64 kB的段长，但不得不允许其他段覆盖，因为实际使用中经常出现代码段远小于64 kB的情形，为了不浪费内存，只好允许覆盖。

保护模式除了段基址之外，还有段限长。这样既兼容了实模式只有一个段寄存器的情况，又相当于增加了一个段尾寄存器。这样既有效地防止了对代码段的覆盖，又防止了代码段自身的访问超限，明显增强了保护作用。

对主奴机制影响深远的是特权级。

从第1章开始，我们多次提到CS的最后两位就是特权级。Intel从硬件上禁止低特权级的代码段的代码使用一些关键性的指令，如LGDT、LLDT、LTR、LIDT。另外，Intel还提供了机会，允许操作系统的设计者通过一些特权级的设置，禁止用户进程使用CLI、STI等对掌控局面至关重要的关键性指令。

有了这些硬件做基础，操作系统就可以把内核设计为最高特权级，把用户进程设计为最低特权级。这样，操作系统设计者就能做到让操作系统内核可以执行一切指令，想做什么就做什么。操作系统可以访问GDT、LDT、TR，而GDT、LDT是逻辑地址形成线性地址的关键，也就是说操作系统能够掌控线性地址，而用户进程则不能。用户进程只能使用逻辑地址，而用户进程的逻辑地址要由内核转换为线性地址。物理地址是由内核将线性地址转换而成的，不知道线性地址就不知道物理地址，也就是说，操作系统内核实际上可以访问任何物理地址。这样，对于用户进程来说，它只能“感觉”到在访问一个逻辑地址的内存空间，如同访问“真实的内存空间”一样，而实际的逻辑地址到物理地址的映射要由操作系统

内核来安排。操作系统把用户进程需要访问的内存安排在内存的什么地方，完全是随心所欲，而用户进程甚至都不知道实际访问的物理地址在哪儿。

Linux 0.11的用户进程线性地址空间的设计方案是，把4 GB的线性地址空间等分为64份，每份64 MB，每个进程一份，每个进程的逻辑地址空间是64 MB。也就是说，在用户进程看来，有64 MB的内存可以使用，这样 $64\text{ MB} \times 64 = 4\text{ GB}$ 。由于用户进程能访问的空间不可能超出64 MB，每个进程的线性地址空间又没有任何交叠，这样，从理论上讲，在线性地址空间的层面，任何用户进程之间的直接相互寻址和访问都是不可能的

了。用户进程之间的相互访问都不可能，更不可能访问操作系统内核了。

这样，保护模式给操作系统的设计者提供了机会，使操作系统的设计者有可能做到，用户进程不能访问操作系统内核，也不能相互访问，而操作系统内核实际上能够访问任何用户进程。这是主奴机制的一个体现。

分页的前提是保护模式，也就是说，**PE**和**PG**必须同时打开，不存在没有**PE**的**PG**。可以说分页和保护是一体的。分页机制同样依托**CPU**的硬件，在提高了内存空间使用效率的同时，也使操作系统的设计者能够实现用户进程之间不能互访，更不可能访问内核，而内核实际上可以任意访问用户进程。

在分页机制下，理论上使线性地址等价于物理地址的分页方法只有一种。下面我们详细讲解这个方法的原理和操作。

我们用一个简单的线性方程来表示线性地址和物理地址之间的关系。

$$y=kx+b$$

这里 x 代表线性地址， y 代表物理地址， k 是线性地址和物理地址的比例关系。因为线性地址和物理地址的单位都是字节，如果增长的方向相同，线性地址和物理地址的比例关系就是1，也就是

$$k=1$$

这样,

$$y=kx+b$$

就成为

$$y=x+b$$

不难看出, 只要

$$b=0$$

就可以实现

$$y=x$$

也就是物理地址等于线性地址。

操作系统内核要想做到在分页机制下实现线性地址等于物理地址，就一定要把操作系统内核分页的起始位置放在物理地址的起始位置，这是关键！

回顾一下第1章的图1-23以及6.2节讲到的内容，才能真正地理解内核分页从内存的起始位置开始的深刻意义，其作用就是 $b=0$ ，使线性地址等于物理地址！

对于内核来说，要直接面对物理内存，最直接的方式就是线性地址和物理地址一一对应。当内核需要访问内存的时候，就可以直接访问物理内存，而不是像进程那样，被操作系统绕过来、绕过去，绕得晕头转向，连自己究竟在物理内存中具体处于什么位置都不知道。

为了在分页机制的前提下满足这项要求，操作系统特别设计了一套页表专门供内核使用。这套页表的值刚好可以使线性地址的值和物理地址的值恒等映射，而且这个映射范围不是局限于内核的1 MB空间本身，而是包含所有16 MB的空间，也就是说，内核实际上可以直接访问任何一个进程的内存空间。具体方案的技术细节已经在第1章图1-39至图1-41以及6.2节详细讲解过。

用户进程只能面对一个逻辑地址，需要使用内存的时候，首先转化为一个线性地址，再根据内核提供的专门为进程设计的分页方案，由MMU转化为实际物理地址。在这里，线性地址和物理地址的转换与直接映射截然相反。

首先，用户进程内存页面的分配是从物理内存的高地址端开始，随着程序的执行向物理内存的低地址端发展的，可以粗略地认为是与线性地址方向反向分配页面，也可以粗略地看成是 $k=-1$ ，而且 b 自然不可能为零。操作系统完全根据多进程实际运行需要给进程临时分配页面，页面分配在物理内存的什么地方事先完全无法预料，准确地说，就是操作系统内核也无法预料，看上去很像随机分配页面。这里就充分显示了内核这个主子直接操作、管理内存，对全部内存范围有使用管理权限，知道每一个进程的实际内存究竟分配在哪里。同时，也充分体现了用户进程——奴才，不知道自己实际使用的内存在哪里。内核代码及所有进程的物理内存分布完全由内核掌控，实现了主奴机制。

具体技术细节已经在6.3节和6.4节详细讲解过。

图9-2是内核及用户进程分页的原理示意图。

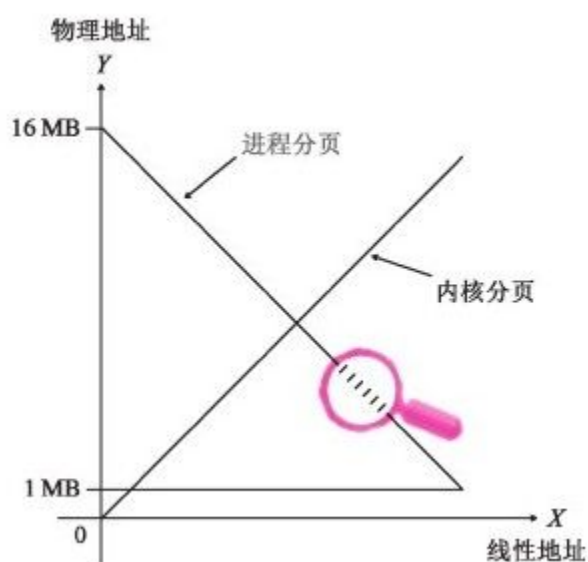


图 9-2 内核及用户进程分页示意图

9.3.2 特权级

特权级主要是依托CPU硬件提供的保护模式，着眼点在“段”，在所有的段选择符的最后两位标识特权级，最终影响的是段选择符决定的段。这些段选择符包括CS、SS、DS、ES、FS和GS。特权级影响的范围是“段”，这是关键点！

对于Linux操作系统而言，通常所说的内核态、用户态，准确的说法是某个代码段、某个数据段或者某个栈段.....当前的特权级或者是0级，也就是内核特权级；或者是3级，也就是用户特权级。

内核特权级可以在任何条件下执行所有的指令。操作系统能够做到用户特权级不能执行的那

些可能颠覆内核特权级的指令。就CPU硬件而言，既可以使所有代码都处于内核特权级，也可以使所有代码都处于用户特权级，如图9-3所示。

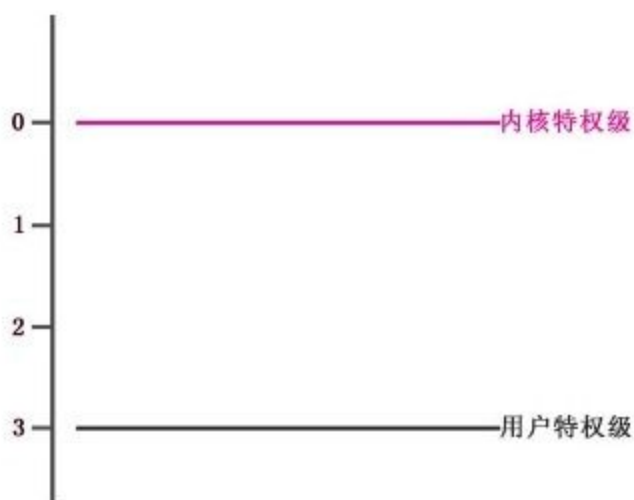


图 9-3 内核与用户特权级示意图

但是，一旦计算机中的所有代码都处于用户特权级，高特权级的指令就永远无法使用了，这也恰恰是特权级设计者的初衷。另外，所有代码都处于同一个特权级，很容易出现互访和覆盖的混乱现象，而且一旦出现混乱就非常麻烦。

操作系统的设计者都是顶级聪明的人，当然不会做这种蠢事。他们会把操作系统内核代码设计为高特权级，把用户进程代码设计为低特权级。这样，内核就可以执行所有的指令，而用户进程则不能，再一次体现了主奴机制。

9.3.3 中断

操作系统和用户程序运行起来后，计算机中的用户特权级和内核特权级的代码、数据频繁交替出现，这是因为有特权级的转换。Intel的CPU提供了几种转换方法，Linux 0.11使用的主要方法是中断。通过中断和中断返回，Linux 0.11实现了特权级之间的转换。图9-4形象地表示了中断所导致的特权级翻转。

下面我们详细讲解为什么中断技术能实现特权级的转换。

在我们看来，计算机中最重要的三件事就是执行序、可识别、可预见。

我们就从执行序开始分析。图9-5是计算机程序执行序的分类示意图。

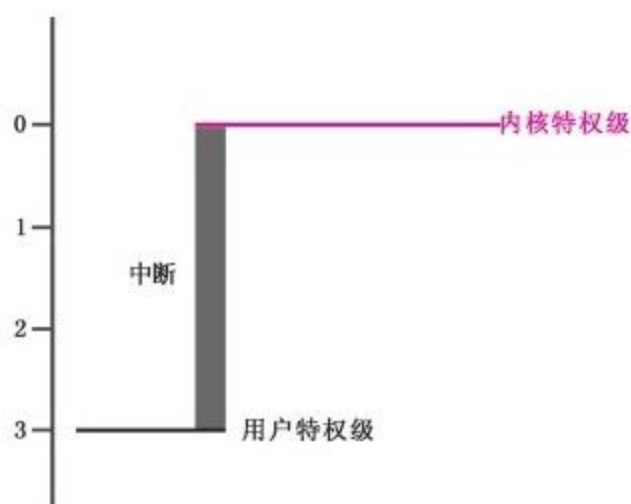


图 9-4 中断导致特权级翻转示意图

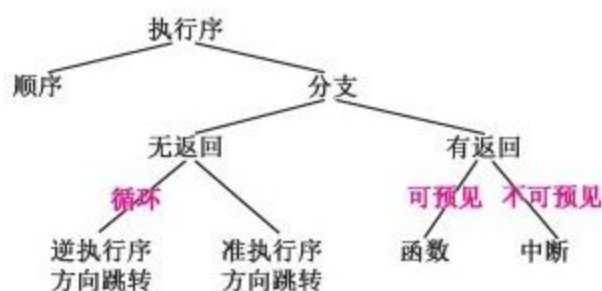


图 9-5 程序执行序的分类示意图

首先，在计算机里的执行序有顺序和分支两种。顺序执行序是靠CPU中的程序计数器PC自动累加实现的。每执行一条指令，PC就自动累加，PC和指令指针IP或EIP联手，形成顺序执行序。除此之外，还有分支，分支又分无返回分支和有返回分支。无返回分支就是跳转，可以通过某种条件进行跳转，跳转以后不会返回。另一类就是跳转之后还要返回，这就是函数调用和中断。更广义的说法是调用子程序。这种情况下执行完子程序的操作之后，还要回到调用指令的下一行继续执行。回到调用指令的下一行的前提是必须满足“能够回到调用指令的下一行”，因此需要保留执行调用指令的状态。这就是所谓的现场保护，实质就是保存标志着CPU和内存运行状态的相关寄存器的值。等到调用结束后，再恢复现场，返

回调用指令的下一行继续执行。从这个角度看，中断和函数有非常像的地方。对程序的设计者而言，二者的差别在于是否可预见，由此导致玩法有重大区别。

函数的调用指令是程序设计者编写的，对程序设计者是可预见的，对于函数调用的保护动作也是可预见的。中断技术最初是因为解决外设硬件的IO问题而发明的，后来又出现了模仿硬件中断的技术路线发明的软中断，方法是类似的。简而言之，中断对于操作系统的设计者来说是不可预见的，随时可能会有一个新的事件在原有的执行序不可预见的情况下切进来，打断原有的执行序，所以叫“中断”。

由于中断的发生不可预见，保护现场的任务不可能由程序员完成，只能由CPU硬件完成，实际上相当于是“硬件的call”。回想一下第2章讲到的如下代码：

```
//代码路径: kernel/fork.c

int copy_process (int nr,long ebp,long edi,long esi,long gs,long
none,

long ebx,long ecx,long edx,

long fs,long es,long ds,

long eip,long cs,long eflags,long esp,long ss)
```

最后一行参数long eip,long cs,long eflags,long esp,long ss在调用之前根本找不到传参。不论是源代码还是反汇编代码，都无法找到传参的代码，也看不到任何压栈动作或将其他数据变成栈的操

作，然而却能正确运行，实在让人费解，原因就是copy_process的调用源于0x80中断，这5个参数是CPU硬件压栈。注意，这5个参数的顺序与Intel IA-32手册中介绍的CPU硬件压栈顺序完全一致。这就是中断的特征。中断服务程序执行结束，自然也要“硬件的ret”——iret。

不知大家是否发现，上面讲解的中断执行程序隐含着另一个特点，就是中断与普通的call有着很大的不同。普通的call似乎是沿着内存的地址“平着”“滑”到被调用位置。中断则不然，似乎是脱离了内存，通过CPU硬件“翻”到内存中另一处中断服务程序的位置。

普通call的这个特点对于编写一般的程序没有任何问题，但对于编写操作系统这样的底层系统

软件却会带来致命的问题。当用户需要使用内核的系统调用代码时，如果用普通的call就能实现，就等价于用户程序可以随意访问操作系统内核，能访问就有可能修改，甚至可能覆盖。这严重背离了主奴机制，必将导致整个系统混乱。

中断通过CPU硬件“翻”到中断服务程序的特征引起了操作系统设计者的注意。当中断通过CPU的硬件“翻”时，CPU硬件的设计者借此机会让CPU翻转了各个段的特权级，使中断成为用户特权级和内核特权级之间翻转的阶梯。除此之外，中断技术还有一个重要的特征，就是能够让硬件的信号直接切进来。对操作系统的调度而言，最重要的就是时钟中断。

如果没有时钟中断，可靠的进程调度是不可想象的。那样的话，操作系统只能设计成与用户进程协商CPU的使用权，只能等待用户进程自觉自愿把CPU的使用权交还，别无他法。

有了硬件的时钟中断就完全不一样了。时钟中断就像操作系统内核手中象征王权的斧钺，根本不与进程协商，时间一到，二话不说，强行将进程的执行序斩断，夺回CPU的使用权，行使主子的特权，体现主奴机制。

9.4 建立主奴机制的决定性因素—— 先机

到此为止，主奴机制似乎已经表述得很好了。只是有一个问题仍然无法解释：用户程序是程序，操作系统也是程序，用的是同一个CPU、同一套指令集，为什么操作系统内核程序能用的指令，用户程序就不能用呢？答案似乎是内核的特权级比用户程序的特权级高。进一步问：为什么内核程序能获得高特权级，而用户程序就不能呢？

我们认为，关键的点是先机！

计算机开机启动的时候是实模式，实模式没有特权级的概念。这时操作系统内核开始加载。正常情况下，此时不应该有BIOS、操作系统以外的任何程序。当操作系统的启动程序打开PE的时候，特权级状态必须是最高特权级，否则，有一部分指令将永远无法使用。

这是非常关键的时刻！操作系统的设计者就是利用这个最有利的机会，以时间换特权，先霸占所有特权，并充分利用这些特权，创建进程。因为所有的进程都是操作系统直接或间接创建的，所以，操作系统有充分的条件和机会把进程的特权级降低。一旦进程的特权级被降低，就再也无法翻身，除非操作系统程序代码设计错误，误把进程的特权级提上来。显然，操作系统设计

者会仔细审查这类错误，并认真测试，消灭这类错误。假如操作系统的代码没有这类错误，进程一旦被创建，就再也无法获得内核特权级，只能终生为奴。由此可见，掌握先机对操作系统主奴机制的形成有着决定性的作用。

一些恶意程序进入计算机的时机虽然晚于操作系统，但它们会想方设法利用操作系统设计上的一切可以利用的漏洞，变被动为主动，抢占先机。一旦掌握先机，马上获得最高特权级，就可以为所欲为……有一类病毒，就通过这一点，利用操作系统的漏洞，想办法驻留到硬盘的系统引导区，甚至是BIOS。大家根据本书前面讲解的原理，就能理解，BIOS和硬盘的系统引导区程序是先于操作系统进入内存的。这类病毒自然就先于

操作系统进入了内存，一旦它们抢占了先机，获得了最高特权级，操作系统就非常麻烦了。

9.5 软件和硬件的关系

计算机分为主机和外设。主机包括CPU、内存和总线；除了主机之外的硬盘、软驱、光驱、显示器、网卡等都是外设。因为，软件编程无法直接控制总线，所以，我们只关注主机中的CPU和内存。

主机进行的是运算；外设进行的是数据的输入、输出和数据的断电保存。

从根本上讲，用户使用计算机是为了解决用户的运算问题。用户运算的直接体现就是用户应用程序。从操作系统的角度看，运行中的用户应用程序就是用户进程。可以说，用户进程代表了用户的运算。

用户的运算需要外设的支持，首先需要键盘、硬盘、网络这类外设将应用程序及需要处理的数据输入主机进行运算。运算的结果需要显示器、打印机等外设输出，需要硬盘等外设进行断电保存和转移，或者驱动其他设备进行工作。以硬盘为例，操作系统将硬盘上存储的数据映射为文件，这些是我们熟知的。进一步拓展文件的概念，从外设上的数据拓展到外设本身，如键盘、显示器也被拓展为字符设备文件。可以说，文件代表用户使用的外设。

我们先展开讲解进程，再展开讲解文件。

9.5.1 非用户进程——进程0、进程1、shell进程

我们先来思考一个问题：所有的操作系统都要有用户使用界面，就是所谓的shell。Linux 0.11的shell是由shell进程而不是由操作系统内核承担的。shell明显是一个操作系统的功能，为什么不是由内核而是由进程承担？

仔细思考一下，我们会发现，如果Linux操作系统只是用于个人计算机，由内核承担shell似乎也没什么不可以。考虑到Linux在服务器领域有很大的发展空间，服务器的操作系统有多shell的需求，这样看来，内核承担shell不划算，最好是由进程承担。

然而，shell的进程显然不能由普通的用户应用程序承担。比如由一个围棋程序承担shell，显然是不合适的。围棋程序本身是应用程序，需要

shell加载。围棋程序自身成为shell，看上去这个操作系统从始至终一直有一个不可退出的围棋，感觉很怪异。如果允许这样的围棋程序退出，就更可怕了。一旦围棋程序结束退出了，操作系统就再也没有shell了。没有shell的操作系统就是一个不可使用的操作系统，有什么存在价值呢？

由此可见，shell必须是一个专门为操作系统配套设计的特殊进程，从开始接受用户使用到关机，都不应该退出。

shell的本质是用户界面程序，掌控的是显示屏、键盘等，这些都是外设。Linux操作系统的进程创建机制是由父进程创建子进程。由此可推断，shell的父进程必须有使用外设的能力以及可用的外设环境，可以推出父进程就应该是进程1那

样的进程。外设肯定是由主机掌控的，所以，所有进程都必须有在主机中运行的能力，可以推出进程1的父进程就应该是进程0那样的进程。

现在，我们可以看得更清楚，第2、3、4章讲解的进程0、进程1、shell进程，就分别对应主机、外设、特殊的外设——用户界面。这三部分也恰恰是计算机宏观构成的三部分。由此可见这三个进程的划分寓意深刻，如果将三个进程合并为一个进程，就不能清晰地表示这种结构。图9-6表示了进程0、进程1及shell进程所对应的硬件。

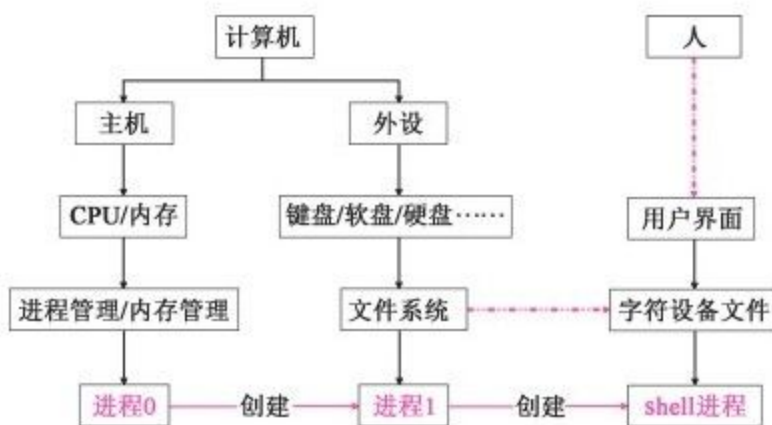


图 9-6 进程与硬件的对应关系示意图

9.5.2 文件与数据存储

从前面章节的内容不难发现，虽然文件系统涉及的代码最多，几乎占了总代码量的一半，但相对而言，文件系统是最容易理解的。以硬盘为例，文件映射硬盘上存储的数据，硬盘的存储空间非常大，远远大于内存的存储量。但说到底，文件也就是一个数据的存储，硬盘可以看做计算机的数据仓库。存储工作虽然手续繁杂，但相对于运算工作而言，要简单得多。而且繁杂的原因主要是硬盘的存储量非常大，而且要“以碎制碎”。如果用简单的管理方法，所需要的管理数据量就很大，而这部分数据既占用了硬盘的空间，又不是用户数据。为了尽可能减少这部分数据在硬盘空间的占有量，用最少的管理数据管理最多

的用户数据，操作系统的设计者提出了超级块、i节点、逻辑块位图、i节点位图等一整套的管理结构。而且这些结构还要涉及进程，导致文件系统变得非常繁杂。但总的来说，文件代表存储的数据（也代表设备），所以相对于运算而言，还是简单得多。

1.内存、硬盘、缓冲区：计算级存储、存储级存储、过渡态

主机中有内存，外设中有硬盘，表面上看，两者的功能都是存储，为什么还分成主机和外设？通常的说法是，内存速度快、价格高、容量小、不能断电保存，硬盘正好是互补的。

进一步追问：如果差别仅仅是断电保存，为什么用两种截然不同的管理方式来管理？操作系统管理内存用的是进程、分页、特权级、表……一大堆复杂的数据结构，管理硬盘用的是文件、i节点、位图、块……差别非常大。

从外形上看，CPU和内存是两种完全不同的器件。实际上，它们必须联合起来才能完成计算机中最重要的工作——计算。也就是说，计算发生在CPU与内存之间，即发生在主机中，仅有CPU是无法完成计算的。CPU计算所需的指令在内存里，计算结果也要放在内存中。不仅如此，更重要的是，复杂的计算，CPU不可能用一个指令完成，需要在内存中安排复杂的算法。例如，把一个复杂的四则运算式转化为逆波兰式后，在

内存中用栈操作的方式形成算法，CPU和内存联合操作，最终取得计算结果。在这个过程中，我们很难否认内存也在进行“计算”，内存有存储功能，更有计算功能，是计算级存储。再看看硬盘，虽然功能也是存储，但丝毫看不出计算的迹象，是非常纯粹的存储，是存储级存储。

计算级存储比存储级存储多了“计算”。计算比存储复杂，管理信息自然要多一些。同样是管理文件，同样是用i节点这套方法，内存中的文件管理信息要比硬盘上的多出一些。硬盘上的文件管理信息就是为了存储，找得到、不出错就行。内存中的文件管理信息就不一样了，除了这些要求外，还要执行查找等操作。查找本身就是计算，所以内存中的管理信息是带有“计算”意义的

文件管理信息。硬盘上的文件管理信息就是一本简单的数据“库存账”。硬盘本身并不执行查找运算，查找运算发生在主机，所以硬盘不需要额外的运算管理信息。

让我们上升一个高度再来看看计算。可以看出这里说的计算也可以分为两类：一类计算是用户进程的计算，也就是用户程序的计算；另一类计算是内核为文件系统的运行所做的计算，这类计算与用户进程的计算没有直接的关系。为了更容易看清楚，我们称参与用户进程计算的内存为全运算存储，称参与内核为文件系统的运行所做运算的内存为半运算存储，称在内存中完全模拟外设、没有任何运算的内存为无运算存储。

有了全运算存储、半运算存储、无运算存储的概念，就容易理解另外一个概念——缓冲区。

缓冲区在内存，是介于全运算存储和无运算存储之间的过渡态。并且，在操作文件的过程中，如在目录文件中查找某个目录项的操作，也是在缓冲区完成的。由于其具体操作是进行字符串比较，肯定是一个运算过程。这些运算明显的不是用户程序的运算，所以缓冲区是半运算存储。

用这个观点审视内存，可以发现内存中的“超级块管理表”和“i节点管理表”（它们常驻于内核数据区）、“逻辑块位图”和“i节点位图”（它们常驻于缓冲区）等文件系统管理结构明显也是服务于半运算的。我们可以把这些管理结构所占的内

存空间（无论是处于内核数据区还是缓冲区）统称为文件系统专用缓冲区。对比通常意义的缓冲区，我们可以看得更清楚，两个缓冲区都由内核控制、操作。通常意义的缓冲区针对的是进程，文件系统专用缓冲区针对的是文件系统。

反过来看，如果没有这两个缓冲区，外设的数据与内存直接交互，操作系统就得在本应是用户进程自己进行计算的全运算存储空间中进行针对文件系统的查找等计算，全运算与半运算搅在一起非常混乱。更麻烦的是，用户进程的全运算源于用户程序的代码，文件系统的半运算源于操作系统内核代码，内核代码和用户代码处理的数据都要在用户进程所属的内存空间操作，有悖于主奴机制。

再者，缓冲区还有一个作用，就是共享。如果缓冲区中有一个进程读入了某个文件，其他进程读同样的文件时，可以共享缓冲区中的文件。没有缓冲区，只能每个进程读自己的文件，这可能造成内存中有同一文件的多份复制。换句话说，缓冲区共享有内存中只有一份复制的含义。

用上述概念考察一下虚拟盘，不难看出虚拟盘就是在内存中简单模拟外设。这类内存空间的特征是无运算存储。这类内存空间映射的不是文件，而是外设，比如软盘。不论这个软盘实际存储的数据有多少，哪怕只有1 kB，也要把整个软盘映射过来，显然浪费了内存空间。又由于虚拟盘是无运算存储，所以用户进程真正使用的时

候，还要经过半运算存储倒一遍手，所以应尽量不用无运算存储。

2.缓冲区的设计指导思想

在设计操作系统的缓冲区时，要求确保多进程读写数据的正确性和尽可能高的效率。内存之间的数据交互速度比内存与硬盘之间的数据交互速度快2~3个量级，缓冲区就在内存中。从数据流向的角度看，缓冲区处在用户进程与硬盘之间。为了实现正确、高效的要求，缓冲区的设计指导思想就是：（1）要使数据的读写有序；

（2）让数据在缓冲区中停留的时间尽可能长。能用缓冲区中的数据，就尽量用缓冲区中的数据。缓冲区中实在没有用户进程所需要的数据了，再从硬盘上读取数据到缓冲区。操作系统中与缓冲

区有关的设计，都直接或间接地体现了这个指导思想。

为了实现这个设计指导思想，Linux 0.11设计了一套数据和相关的函数，这些数据包括hash_table、b_count、b_lock、b_dirt、*b_data、b_dev、b_blocknr、b_uptodate、b_wait.....

用*b_data指向与进程进行数据交互的缓冲块，用b_dev和b_blocknr指定将要读写的文件的一个数据块所在的设备号和块号（以后简称为硬盘块），并将这一对缓冲块、硬盘块挂接在hash_table上，形成绑定关系。依此类推，hash_table将所有需要读写的硬盘块与对应的缓冲块绑定，形成如图9-7所示的管理关系。

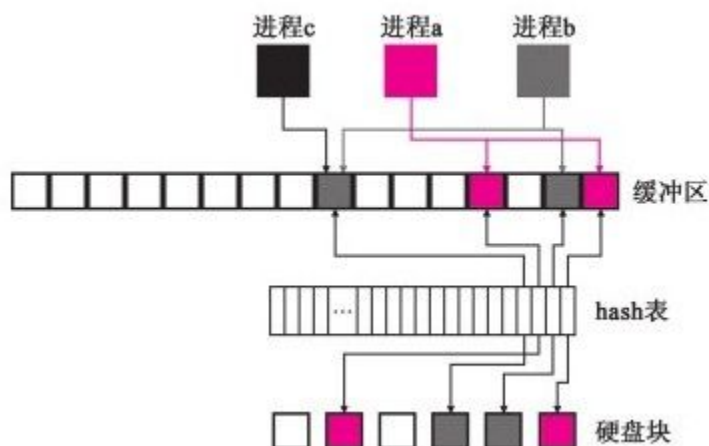


图 9-7 缓冲区、哈希表与硬盘块的管理对应关系示意图

当用户进程要进行文件读写操作时，并不一定是对文件的所有数据都要读写，操作系统通过对文件的分析，确定要操作哪个硬盘块，用**b_dev**和**b_blocknr**标识。为了最大限度地提高缓冲块数据的复用性，此前的文件读写任务完成后，与硬盘块对应的缓冲块中的数据不会立即被清除掉。所以，执行新的文件读写任务时，操作系统会先在缓冲区的管理结构**hash_table**中查找，看看记载

的和缓冲块对应的硬盘块与操作系统确定需要读写的硬盘块是否相吻合，只要相吻合，就证明需要操作的硬盘块的数据，很有可能不需要读盘了。根据“能用缓冲区中的数据，就尽量用缓冲区中的数据”这个指导思想，能用现成的数据就用现成的。

然而，`b_dev`和`b_blocknr`这两个字段吻合，只能说明缓冲区中有操作系统需要操作的硬盘块对应的缓冲块，并不等于缓冲块中的数据一定可以使用，因为这个缓冲块中的数据有可能已经无效。比如说某个文件的内容已经全部被删除，那么这个文件“残存”的缓冲块中的数据还在，缓冲块与硬盘块的绑定关系也在，但已经不能再使用了。

为了解决这个问题，操作系统设置了一个 `b_uptodate`。当它的值为1时，说明缓冲块中的数据是有效的，可以直接使用，不用再次读盘了；如果为0，说明该缓冲块中的数据无效，不能直接使用，还需要从硬盘块中读出新的数据，缓冲块才能使用。`b_uptodate`对于读操作尤为重要。在硬盘块中的数据读入缓冲块后，操作系统的中断服务程序就把**`b_uptodate`**设置为1，表示该缓冲块中的数据此时已经有效。在新申请一个缓冲块时，`b_uptodate`就会被设置为0，表示缓冲块中的数据无效。

在操作系统设计者看来，用户进程将数据写盘，其实就是操作系统将用户进程的数据写入缓冲区。缓冲区中的数据什么时候真正写到硬盘上

去，由操作系统决定。换句话说，用户进程的数据写盘是分两步进行的：第一步，操作系统将用户数据写入缓冲区，并尽可能停留在缓冲区，尽可能复用；第二步，操作系统适时写盘。这两步通常不是连续执行的，中间可能会有一个停顿。为了让操作系统在停顿后仍能够高效地同步缓冲区中的数据，避免不必要的同步，操作系统设计者设计了**b_dirt**。其作用就是标识此前操作系统曾将用户进程的数据写入缓冲块中，改变了缓冲块中的数据。将其值设置为1，表示所管理的缓冲块中的数据需要同步到硬盘上。等到操作系统将缓冲块中的数据同步到硬盘后，把该字段改为0。

注意，写操作和读操作还不太一样。如果是读操作，而且缓冲块中没有现成的数据，就只有

从硬盘块中进行读取，而且是立即读取，因为用户还在等待使用这些数据。写操作则不同，用户进程并不知道所谓的写盘其实只是将数据写到缓冲块中，缓冲块中的数据什么时候同步到硬盘，完全由操作系统酌情而定。在开始同步工作前，即使**b_dirt**已经为1，如还需要往这个缓冲块中继续写入数据，仍然可行。将来操作系统只同步最终的数据，这也是缓冲区设计指导思想的另一种体现。

为了保证数据读写的正确性，必须确保数据读写的有序。比如，不能在缓冲块数据同步到硬盘块的同时，还在向这个缓冲块中写新的数据。**b_lock**就起这个标识作用。在同步缓冲块之前，必须把这个缓冲块加锁，即**b_lock=1**。操作系统

内核见到这个标志，就不再将进程的数据写入这个缓冲块中。这样在同步的过程中，缓冲块中的数据不会发生任何变化，确保同步结束时，缓冲块与对应的硬盘块中的数据是一致的。也就是说读写缓冲块与同步数据这两步操作不能同时进行。当**b_lock**设置为0时，只允许操作系统进行进程与缓冲块间的数据交互；当**b_lock**设置为1时，只允许操作系统进行硬盘与缓冲块间的数据交互。这样就能避免同时操作。

当缓冲块被加锁时，仍有其他进程需要与被加锁的缓冲块进行数据交互的可能。由于此时操作系统禁止任何进程与被加锁的缓冲块交互数据，所以操作系统只有将这样的进程挂起，切换到其他进程去执行，并用***b_wait**指向被挂起的进

程，以便缓冲块解锁后唤醒被挂起的进程。当需要与被加锁的缓冲块进行数据交互的进程不止一个时，`*b_wait`实际上指向的是最后一个申请与被加锁的缓冲块进行数据交互的进程，其余的进程按序形成一个隐含的队列，如图9-8所示。

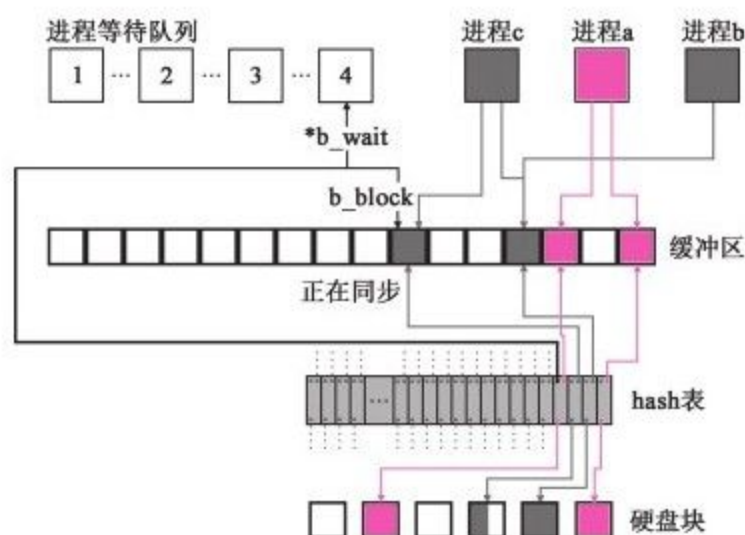


图 9-8 多进程访问设备文件时的状态示意图

当一个进程需要与硬盘进行数据交互时，操作系统首先遍历缓冲区的管理结构哈希表。如果

通过哈希表找到现成的缓冲块，哪怕它还被其他进程使用着，只要数据有效，就先用这个现成的。用现成的就不用从硬盘块中读取数据，比操作硬盘划算得多。实在是找不到现成的，再申请一个空闲的缓冲块。`b_count`就是缓冲块是否在空闲状态的标志。事实上，可能有多个进程提出申请，需要与同一个缓冲块进行数据交互，每增加一个提出申请的进程，`b_count`就加1，反之就减1。如果`b_count`最后被削减为0，说明这个缓冲块没有被引用，它就是空闲缓冲块。

有了这些措施，就从体系上保证了用户进程所要读写的数据与硬盘对应数据的一致性，并且实现了尽可能的高效。

下面我们讲解一下文件系统和进程的另一个连接点——管道。

3.利用文件系统实现进程间通信——管道

管道是进程间通信的一种方式，位于内存，是内存中的概念。奇怪的是，管道不是用内存管理的方式进行管理的，而是用文件系统的管理方式来管理的。这是为什么？

进程管理的设计指导思想，就是要使进程之间完全独立和隔离。保护模式就是根据这个要求设计的。进程间通信意味着数据要跨越进程边界流动，如果采取直接交互的方式，显然违背了这个设计指导思想。怎样做才能使数据合理地跨越

进程的保护边界进行流动，既实现进程间通信，又不破坏操作系统对进程边界的保护？

仔细分析，我们就会发现对于进程来说，文件是一种每个进程都可以访问的资源。换句话说，文件对进程而言是共享的。如果将进程间需要通信的数据以文件作为中转站，多个进程同时访问一个文件，有的进程写数据、有的进程读数据，就可以实现进程间彼此的数据传输。这么做既满足了进程间独立、隔离的基本思想，又实现了进程之间通信的功能。

然而，文件代表的是外设，CPU与外设的通信速度比CPU与内存的通信速度慢2~3个量级。操作系统既然可以在内存中虚拟软盘，肯定也可以虚拟文件。在内存中虚拟一个专门用作进程间

通信的文件，这就是管道。操作系统有了管道，就实现了既以文件作为进程间通信的中转站，又享受内存级通信的速度。由于管道来源于文件，管理方式自然与文件相同。这就是管道为什么通过文件系统管理。

9.6 父子进程共享页面

当父进程创建子进程的时候，操作系统先将父进程的全部管理数据结构复制给子进程，在子进程加载自身的代码之前，分享父进程的代码，等到加载自己的代码后，才切断共享父进程代码的关系。为什么不在子进程刚刚创建完毕就切断这个关系？

因为此时的子进程没有任何自己的代码，加载自己代码的工作也是需要代码的，这份代码按照Linux的规则，只在父进程中有，所以如果不能共享父进程的代码，就连加载自身代码的工作也无法完成。

共享父进程代码的机制为很多服务器程序提供了方便。既然允许共享，就应该允许子进程实实在在地执行父进程的代码，这样就面临父子进程同时使用同样的代码、数据的局面，极有可能造成数据混乱。为了避免这种情况的发生，顺理成章地提出页写保护机制。具体技术细节在6.4.2节讲解过。页写保护机制的设计指导思想是防止多进程访问共享数据导致的数据混乱。依此类推，对所有访问共享数据（包括内存的、外设的）可能导致的数据混乱，解决问题的基本思路与此类似。

9.7 操作系统的全局中断与进程的局部中断——信号

我们在前面已经反复多次提到中断。中断对操作系统而言，其重要性怎么强调都不过分。下面，我们沿着中断的技术路线向前延伸，分析中断与`task_interruptible`、`task_uninterruptible`之间的关系。

在本书的第1章就讲到过CLI关中断，我们知道CLI可以使整个操作系统不能收到中断信号，等于关了整个操作系统的中断。

`task_interruptible`和`task_uninterruptible`，虽然它们名字里含有中断字样，却看不出它们和通常

意义上的中断有什么直接的关系。

追踪task_interruptible和task_uninterruptible的使用，可以发现与它们关系紧密的是信号。为什么和信号有关系的参数却起了一个和中断有关的名字？

回顾中断技术可以知道，中断技术发明的最初动机是为了避免操作系统频繁地主动轮询外设的IO状态，空耗主机资源。中断技术使操作系统由主动轮询变为被动响应，极大地降低了IO过程中主机资源的消耗，提高了运行效率。

对比分析中断和信号，可以看出信号明显是在模仿中断的技术路线，使进程间的沟通由主动轮询变为被动响应，同样大幅度减少了进程间沟

通引起的操作系统的消耗，提高了整体运行效率。比如，`shell`进程创建了一个子进程，如果子进程退出，理论上，应该由`shell`释放子进程的进程管理结构，为子进程的退出做善后工作。问题是：`shell`怎么知道子进程要退出呢？很容易想到的方法是询问子进程是否退出。如果`shell`创建了几十个子进程，依照这个方法，就要对每一个子进程进行定期轮询，查看是否有子进程退出，而且不论有几个子进程要退出，即使一个要退出的子进程都没有，`shell`为了及时处理子进程的退出，仍然要频繁地轮询子进程。这与中断技术发明之前，主机频繁轮询外设的IO情形极其类似。操作系统的设计者借用中断的技术路线，设计了模拟中断的信号。我们对比一下就会发现，两者

的技术路线非常相似。两者的对应关系如图9-9所示。

中断	信号
cli	TASK_UNINTERRUPTIBLE
sti	TASK_INTERRUPTIBLE
IDT、中断向量	sigaction
中断服务程序	信号服务程序
*****	*****

图 9-9 中断与信号的对比示意图

可以发现，两者的对称性很强，有很强的可比性。两者的区别在于中断是针对整个操作系统的，而信号是针对进程的。我们甚至可以把通常的中断看成整个操作系统的“全局中断”，信号是进程的“局部中断”。这样可以更清楚地看到信号

的本质及精髓。读者可以利用对比的方法，深入理解、掌握信号以及`task_uninterruptible`、`task_interruptible`。

9.8 本章小结

到此为止，我们已经把操作系统的设计指导思想这一章的内容讲解完毕。当然，一个操作系统是非常复杂的，要想设计出一个可用的操作系统，仅凭这一章的内容是远远不够的。但本章的内容对帮助读者站在操作系统设计者的角度全面理解、把握操作系统仍然是充分的。

结束语

现在是全书的结尾，很高兴能在此和您见面。根据我们多年的教学经验，能在这里和您见面，说明您在操作系统方面的水平已经不可小视！因为操作系统实在是太难了，绝大部分读者中途就已经放弃了。如果您还感到意犹未尽，就请您回过头去重新阅读，细细品味！

“新设计团队”简介

本书由中科院的指导老师杨力祥先生带领的“新设计团队”共同完成。

作者的话

梁文峰



操作系统的源代码错综复杂，千头万绪，但在我看来，操作系统并不是一堆源代码，而是一

部精密的机器。看代码之前，眼前会先浮现出一部正在运转的机器，而代码提供的就是这部机器的“设计图纸”，它说明了：这部机器启动过程中要打开哪些开关，以及它们打开和关闭的先后顺序和相互之间的影响等。随着了解的不断深入，整部机器的架构和设计的细节在脑海中变得越来越清晰，在此过程中，有时候会发现自己对机器的某个部件的运转情况不太清楚，于是就会根据自己对机器情况的了解推测出其他应该会有与该部件的运转相配合的其他部件，于是拿出“设计图纸”对照，结果与推测的完全一致。

在了解这台机器的过程中，也会不断思考一些问题：比如，各零部件为什么要这样设计，为什么要以这样的顺序启动，启动后为什么要这样

响应某些操作，更重要的是，这一切都是唯一的吗，有没有其他可能，后果又会是什么？

在我的导师杨力祥老师的帮助下，我不仅对操作系统中“是什么”有了全面的认识，而且对“为什么”有了体系性的提高。在我看来，理解“操作系统”这种机器的原理的过程与理解“汽车”、“坦克”、“战斗机”等这些机器的原理的过程在本质都是一样的。

本书是我们对这部机器的运转情况及其本质原理的生动描述，愿与您分享。

陈大钊



用“图”说话，这是本书的最大特色。的确，当你试图去描述某种有着复杂结构的事物时，图形远比文字更能让人理解。

有时候说了半天，还真不如画个图来得简单。书中包括339张图，每张图都是精心设计和绘制的，每条线的像素宽度和位置都是经过精确计算的。毫不夸张地说，这本书不仅是在讲解操作系统，更是在绘制操作系统！本着对读者负责的态度，本书在所有技术细节上都力求完美！

刘天厚



我很幸运在杨老师的指引下加入了“新设计团队”，又很幸运地与新设计团队的伙伴们一起共同努力完成了本书的创作。虽然操作系统是电脑中最最复杂的程序，几乎所有讲解操作系统的书都让人望而生畏，但是本书不是一本让你只看几页就看不下去并束之高阁的书，它极有可能是一本会被你翻烂的书。我们以抽丝剥茧的方式对Linux操作系统的内核设计原理进行了剖析，希望能帮

助大家从宏观到微观上去认识整个操作系统。虽然99%的程序员一辈子也不会对操作系统的架构和设计进行修改，但是我们希望每个程序员都能从本书中或多或少地得到一些启发，成为那1%。

冯克



操作系统在计算机中的地位举足轻重，但是它的学习难度也是非常大的。很多人学习操作系统的信心和决心，常常会被各种学习资料中密密麻麻的文字和庞杂的程序代码而击溃。

我所在的“新设计团队”在研究Linux操作系统时以它的真实的运行时序为主线，从一个全新的视角对操作系统进行了重新审视，于是我们的研究在短时间内取得了显著的成果。

我们在研究的过程中以图解的方式勾勒出了操作系统的真实运行时序，并根据这种思路完成了本书。通过书中精心绘制的、能真实反映操作系统运行时序的300多幅图，即使你不看操作系统的源代码，也可对操作系统的运行有初步的了解。当你深入分析每一幅图的细节时，结合与图相关的说明文字，你将会发现操作系统的原理与机制与真实的运行代码是无缝衔接的。如果理解了本书中的内容，操作系统不将再是虚无缥缈

的，它的每一个动作都能在源代码中找到相应的实现。

希望此书的与众不同的讲解方式能给你带来全新的学习体验，希望它能帮你真正学懂操作系统，少走弯路。

武若冰



不懂汇编语言和C语言能看懂操作系统的源代码吗？你可能认为我在开玩笑，不过这对我来说

说并不是玩笑，而是事实。我是环境工程/环境化学科班出身，连谭浩强先生的入门级C语言教程都不曾读过。一年多前，当杨老师说让我和大家一起研究和学习操作系统时，那时我也曾彷徨过。好在杨老师的方法很独特，在他的指导下，我很快就入了门。书中的内容是我学习操作系统时的学习方法和思维方式的真实写照。

当你手捧这本书时，看过我的学习经历之后，你是不是对自己学习操作系统更加充满信心？本书独特的讲述方式、精心计算与绘制出的图片，以及对操作系统设计原理的精辟的讲解，将会成为你操作系统探秘之旅的指南针。

宋琦



这本书里面有很多内容都是其他同类书中不曾探讨的，比如：操作系统是如何进入怠速过程的？`main`函数竟然不是操作系统的起始点？内核程序 and 用户程序的等级为何如此森严？操作系统中居然还有“主奴机制”？这些结论不但生动有趣，而且对于理解操作系统有重大的意义。

然而，这并不仅是一本描述结论的书。本书真正的精华是它呈现出的一种体系性的思考方式，这种思考方式指导我们在纷繁复杂的操作系

统源代码中理清了它的核心设计思想，然后我们根据这种思想构建了自己的理论体系，以指导实际工作和解决具体问题。这种思考方式不仅适用于操作系统，而且它是一切原发性思考的源动力。这种思考方式是对几十年来我们只学科学结论却忽略科学探索方式的填鸭式教育的突破。对操作系统的理解只是本书的作者们在这种思考方式下解决具体问题的一个很小的案例。这段经历犹如一段美妙的旅程，是一个充满了推论、探索、假设、证伪、构建体系的过程。如果你现在已经准备好了与我们一起分享这个过程，那么，祝你旅途愉快！

联系作者

如果你对本书有意见或建议，欢迎你通过邮箱：xsjlinuxOS@163.com与作者联系。